VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace automatických aktualizací v jazyce JAVA

Automatic Update for JAVA applications

2010 **Pavel Piskoř**

Zadání bakalářské práce

Student: **Pavel Piskoř**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace automatických aktualizací v jazyce JAVA

Automatic Update for JAVA application

Zásady pro vypracování:

Cílem práce je navrhnout a naimplementovat systém automatických aktualizací pro aplikace v jazyce JAVA, který umožní těmto aplikacím po spuštění automaticky zkontrolovat dostupnost nových verzí na serveru.
Systém musí splňovat tyto požadavky:
1. Možnost aktualizace aplikace a stažení pouze těch JAR souborů, které jsou aktualizovány.
2. Možnost kontroly licenčních klíčů k aktualizaci aplikace.
Práce bude obsahovat:
3. Přehled současného stavu v oblastí nástrojů pro automatické aktualizace.
4. Návrh systému pro automatické aktualizace.
5. Implementace systému pro automatické aktualizace (serverová i klientská část) a jeho použití pro program UniSave.

Seznam doporučené odborné literatury:
Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: Ing. David Ježek, Ph.D.

Datum zadání: 30.11.2008
Datum odevzdání: 07.05.2010

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Podpis a datum odevzdání bakalářské práce …….....

Poděkování

Děkuji tímto mému vedoucímu bakalářské práce Ing. Davidu Ježkovi, Ph.D. za odborné vedení této bakalářské práce.

Abstrakt

Cílem této bakalářské práce je poskytnout jednoduchý úvod do oblasti automatických aktualizací, přehled současných nástrojů podporujících implementaci automatických aktualizací a také navrhnout a naimplementovat jedno z možných řešení automatických aktualizací s předvedením jeho použití v praxi. Návrh a implementace řešení se omezuje pouze na aplikace napsané v jazyce Java.

Klíčová slova

Automatické aktualizace, Java, JVM, zaváděč tříd, zavádění tříd

Abstract

The aim of this bachelor thesis is to provide the basic introduction to automatic update domain, overview of current tools that supporting implementation of automatic update and also to draw up and to implement one of the possible solution of automatic updates with demonstration its using in work experience. The draw and the implementation of this solution are restricted only to application written in Java language.

Keywords

Automatic updates, Java, JVM, class loader, class loading

Seznam použitých symbolů a zkratek

Java – programovací jazyk

Java aplikace – program napsaný v jazyce Java

Java API – (Java Application Programming Interface) – základní kolekce tříd pro v jazyce Java

JVM – (Java Virtual Machine) – Virtuální stroj jazyka Java, umožňující běh aplikací napsaných v tomto jazyce.

JAR – (Java archive) – kompresní souborový formát, založený na kompresi ZIP.

HTTP – (Hypertext Transfer Protokol) – internetový protokol sloužící, pro přenos hypertextových dokumentů.

XML – (Extensible Markup Language) – obecný značkovací jazyk, určený pro výměnu zpráv mezi aplikacemi, pro publikaci dokumentů nebo sloužící pro tvorbu jiných jazyků.

XSD – (Schema Definition) – XML schéma, které popisuje strukturu dokumentu XML. Alternativa k staršímu typu schématu DTD.

SOAP – (Simple Object Access Protocol) – protokol, pro výměnu zpráv založený na XML formátu. Většinou používá pro přenos HTTP protokol.

JAXB – (Java Architecture for XML Binding) – technologie pro práci s XML strukturou v jazyce Java. Převádí textový formát XML na Java objekty a naopak.

Seznam použitých obrázků

Obsah

1 Úvod

Většina výrobců softwarových děl, potřebuje čas od času své dílo aktualizovat. Důvodů se najde celá řada. Může to být například oprava chyb, které se vyskytnou během vývoje softwarového díla. Žádné větší softwarové dílo se totiž neobejde bez chyb. Před předáním hotového softwarového díla zákazníkovi sice proběhne řada testů, které spoustu chyb odchytí, ale ne vždy se podaří odchytit všechny chyby. Tak se stává, že při používání hotového díla zákazníkem, se na některé tyto chyby příjde později a vznikne potřeba tyto chyby odstranit. V mnoha případech je nemožné doručit takovou opravu každému zákazníkovi zvlášť, případně obesílat zákazníky poštou či e-mailem o dostupné opravě. Proto se volí snadnější způsob opravy softwarových děl a tím je možnost aktualizace daného software přes síť internet. Dalším důvodem k aktualizaci může být přidání nové funkcionality do již používané aplikace nebo změna stávající funkcionality. Zákazníkovi například vznikne potřeba rozšířit stávající aplikaci o nějaký další modul. Ze strany výrobce zase může jít o vylepšení nějaké části aplikace, které danou aplikaci zrychlí, zpřesní její činnost nebo jinak zdokonalí. Třetím důvodem pro provádění změn v aplikacích může být snaha o udržení některých částí aplikace v souladu s aktuálními podmínkami a požadavky zákazníka. Jedná se o aplikace provádějící například účetní nebo celní operace, které se mění podle aktuálního daňového či celního zákona. Patří zde i antivirové programy, které musí být neustále rozšiřovány o informace o nově vzniklých problémech.

Nejen výše zmíněné záležitosti ale i mnoho jiných se dá spolehlivě řešit takovými aplikacemi, které umožňují svou aktualizaci ať už manuálně či automaticky. Aktualizací je zde myšleno nahrazení určité části aplikace novou části. Pokud taková aplikace navíc provádí své aktualizace zcela automaticky, značně ulehčí svému uživateli práci a čas. Některé aplikace disponují možnosti aktualizace již od počátku, protože už při návrhu se na tuto funkcionalitu myslelo. Existuje ale celá řada takových aplikací, které tuto možnost nemají. Jejich autoři, pokud to okolnosti vyžadují, pak mají na výběr buďto doimplementovat celou tuto chybějící funkcionalitu svými silami nebo využít již existujícího rámce či knihovny s touto funkcionalitou, pro které pak jen mírně přizpůsobí svojí aplikaci.

Táto práce se tedy zaměřuje ve své druhé kapitole na uvedení do problematiky automatických aktualizací, objasnění jednotlivých principů a postupů při provádění aktualizací a rozděluje aktualizace do několika oblastí použití. Třetí kapitola pak poskytuje přehled některých současných nástrojů pro podporu vývoje software obsahujícího modul automatické aktualizace. Ve čtvrté kapitole se objevuje návrh jednoho z možných řešení automatických aktualizací. Návrh obsahuje velmi jednoduché, ale plně automatické řešení, které by se dalo v mnoha směrech vylepšit. Pátá kapitola ukazuje implementaci takového řešení v jazyce Java s použitím několika dalších technologií. Šestá kapitola pak ukazuje jednu z několika možností, jak takové řešení použít v konkrétní aplikaci. Je zde předvedeno nalezení vhodných míst v již existující aplikaci pro rozšíření o modul automatických aktualizací a logika prováděných kroků pro různě případy které mohou nastat při samotné aktualizaci.

2 Problematika aktualizací

Táto kapitola pojednává o základní problematice aktualizací aplikací. Nastiňuje jak obecné principy této problematiky, tak také konkrétní, z pohledu programovacího jazyka Java. Je jakýmsi úvodem do této oblasti a slouží pro lepší pochopení dalších částí této práce.

2.1 Aktualizace obecně

2.1.1 Rozdělení aplikací podle dostupnosti

Jak plyne z úvodní kapitoly, je spousta aplikací, pro které je možnost aktualizovat se nezbytnou součástí. Některé z těchto aplikací se používají jen občasně a po spuštění jsou v provozu řekněme pár hodin. Například konvertor audio či video formátů. Jiné se mohou používat častěji nebo mohou po spuštění vykonávat časově náročné úkoly, které zaberou několik hodin případně dnů nepřetržité práce. Například aplikace pro různé vědecké výpočty nebo tvorbu kreslených filmů. A pak máme aplikace, jejichž provoz je vyžadován po spuštění trvale. Jsou to především bankovní systémy, systémy pro elektronické obchodování a podobně.

Rozdělme si tedy aplikace z hlediska dostupnosti na ty, které by měly běžet trvale a jejich přerušení je nežádoucí a na ty, kterým přerušení běhu nezpůsobuje žádné problémy. Toto rozdělení aplikací je důležité jak při výběru vhodného rámce či knihovny umožňující aktualizační proces, tak při vytváření vlastního řešení. Je to z toho důvodu, že po aktualizačním procesu je ve většině případů nutno aplikaci ukončit a po té znovu spustit. Ukončení a znovu spuštění, tak zvaný „restart“ je potřebný buď pro zamezení možných problémů při změně části kódu nebo proto aby se změny mohly projevit.

Než se pustím do dalšího výkladu, upřesním ještě význam slova „přerušení“. Každý proces aktualizace zabere určitý čas, po který je činnost aplikace pozastavena. Toto pozastavení zde nebudeme brát jako přerušení aplikace. Jako přerušení budeme brát situaci, kdy je potřeba aplikaci zastavit a po té znovu spustit. Táto situace totiž ve většině případů vyžaduje přítomnost člověka jako obsluhy.

2.1.2 Rozdělení aplikací podle „pamatování“ si stavu nebo dat

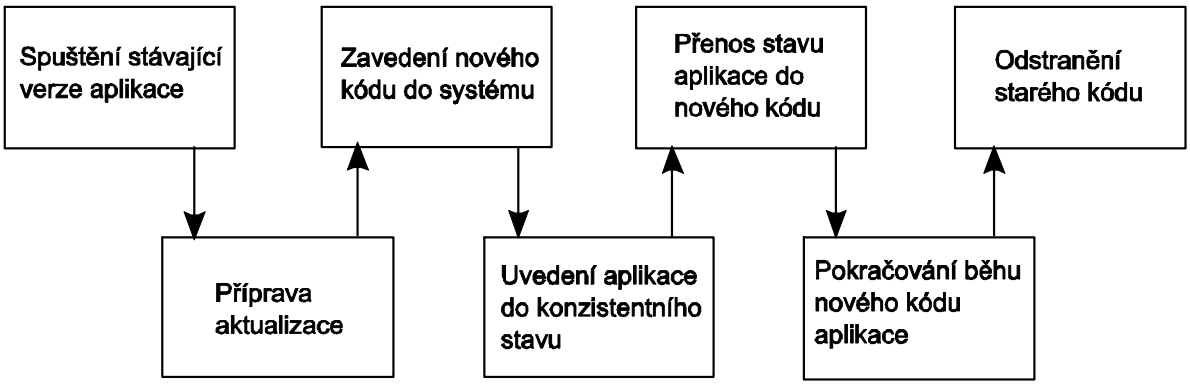
Další hledisko, které hraje určitou roli při aktualizacích, je to jestli aplikace obsahuje pouze výkonnou část (metody) nebo obsahuje navíc i nějaká data, které je třeba uchovat po celou dobu běhu aplikace, případně uchovává stavy, ve kterých se aplikace právě nachází. Mějme příklad aplikace, která se chová jako služba poskytující určité činnosti. Použití takové aplikace spočívá v tom, že jí „předáme“ nějaká data a „řekneme“ co s těmi daty požadujeme udělat. Aplikace provede požadovanou činnost na těmito daty a vrátí nám nějaký výsledek. Po té čeká opět na další požadavek. Taková aplikace nebo případně jen určitý modul aplikace nemá žádné specifické vnitřní stavy, které by si bylo nutno pamatovat ani neuchovává žádná data. Jiným případem bude aplikace, která obsahuje třídy představující datovou strukturu. V objektech vzniklých z těchto tříd pak taková aplikace uchovává nějaká data, se kterými pracuje po celou dobu svého běhu. Jiný typ aplikací může obsahovat procesy, které zpracovávají nějaká data v jednotlivých krocích sekvenčně podle předem daných předpisů a procházejí přitom určitými stavy, které jsou aplikací rozlišovány. Zde je potřeba si po celou dobu běhu aplikace pamatovat u každého procesu stav, ve kterém se daný proces zrovna nachází.

Rozdělme si tedy aplikace z hlediska pamatování si na ty, které mají potřebu si něco pamatovat a na ty, které si nemusí nic pamatovat. Je to z toho důvodu, že v některých případech aktualizace je potřeba

rozhodnout, zda se bude přenášet stav staré verze aplikace do nové verze aby nedošlo ke ztrátě určitých informací, které by pak mohly mít negativní dopad na chování aplikace po aktualizaci.

2.1.3 Základní kroky aktualizace

Celý proces aktualizace se dá rozdělit do několika obecných kroků jak je vidět na obrázku 2. Ne každá aplikace bude obsahovat všechny tyto kroky při své aktualizaci. První tři kroky se však vykonávají ve všech případech aktualizací.



Obr. 1. Jednotlivé kroky při aktualizaci

První krok, spuštění aplikace. Nejdříve je potřeba aplikaci spustit. V případě velkých systému se jedná o jejich nasazení.

Druhý krok, příprava aktualizace. Příprava aktualizace zahrnuje mimo jiné například spojení se serverem, činností pro rozpoznání dostupnosti nové verze aplikace a stažení nových souborů. Při spojování se serverem může být volitelně provedeno ověřování licenčních klíčů pro přístup k aktualizacím nebo jiný způsob ověřování přístupu v případě placených nebo jinak přístupově omezených aktualizací. Při stáhnutí souborů pak může, ale nemusí být provedena kontrola na důvěryhodnost souborů z hlediska bezpečnosti, zabraňující podvrhnutí nepravých souborů do aplikace.

Důležitou činností je při přípravě aktualizace dosažení tak zvaného „bezpečného stavu“ aplikace. Bezpečným stavem aplikace rozumíme stav, kdy aplikace neprovádí „žádnou“ činnost, při které by proces aktualizace mohl aplikaci negativně ovlivnit. Negativním ovlivněním je myšleno například porušení nebo ztráta dat, narušení nějakého výpočetního procesu, který následkem toho vede ke špatnému výsledku a podobně. V podstatě to znamená, že každá manipulace s daty prováděná současně s procesem aktualizace by mohla vést k neočekávanému chování aplikace. Ve skutečnosti ovšem záleží také na tom, která část aplikace je aktualizaci ovlivněna. Pokud je aplikace „vhodně“ navržena, například se skládá z několika navzájem se neovlivňujících modulů, je možné aby některý z modulu prováděl svou činnost, zatím co jiný je aktualizován.

Dosažení bezpečného stavu aplikace lze docílit například volbou vhodného okamžiku pro provedení aktualizačních procesů. Nejjednodušší je umístit aktualizaci na samotný „začátek“ aplikace, kdy ještě nedošlo jejímu plnému rozběhnutí nebo na konec aplikace, těsně před jejím ukončením, kdy už se nebude používat žádná část kódu.

Další možností je zavést v aplikaci tak zvaný „příznak aktualizace“, ten aktivovat a počkat, až se aktualizace do bezpečného stavu dostane. Jako příznak aktualizace může sloužit proměnná typu

boolean, která se při potřebě aktualizovat aplikaci aktivuje (nastaví na hodnotu „true“). Jaká koli činnost v aplikaci (metoda, procedura či proces) by před svým provedením kontrolovala tento příznak a pokud by byl příznak aktivován, odložila by aplikace její spuštění až do doby deaktivace tohoto příznaku. Čekání na bezpečný stav aplikace pak může spočívat ve sledování všech běžících metod, procedur či procesů dokud neskončí. Po aktualizaci se pak příznak deaktivuje a aplikace běží dál nebo se ukončí a znovu spustí (restartuje).

Třetí krok, zavedení nového kódu do systému. Spočívá v nakopírování nových souborů do složky s aplikací a případném propojení se stávajícím kódem. Některé programovací jazyky umožňují jednoduše provádět změny v kódu za běhu aplikace, jiné to neumožňují. V případě jazyka Java není změna v kódu za běhu aplikace zrovna jednoduchá. Z tohoto důvodu se většinou volí restartování aplikace po provedení aktualizace. Pokud aplikaci restartovat nechceme nebo nemůžeme je potřeba nejdříve vytvořit všechny potřebné objekty nových verzí tříd a po té „přesměrovat“ všechny reference ukazující na staré verze objektů tak, aby nyní ukazovaly na nově vytvořené objekty.

Čtvrtý krok, uvedení aplikace do konzistentního stavu, již není obsažen ve všech aktualizacích. Dosažení konzistentního stavu aplikace je vyžadováno v případě kdy není možné aplikaci přerušit v jejím běhu. To znamená že aktualizace je prováděna za provozu aplikace bez možnosti restartu. Zajištění konzistentního stavu zamezí vzniku neočekávaného chování aplikace a případného porušení dat po aktualizaci. Tento úkol v sobě obsahuje také již zmíněnou aktualizaci referencí na nově vytvořené objekty aplikace. Pokud by se totiž ponechala nějaká reference na starou verzi objektu a zároveň by existovala i reference na verzi novou nebylo by zajištěno jednoznačné chování aplikace. Část kódu by se mohla chovat starým způsobem a část novým. V případě, že by se naopak odstranila reference na starý objekt a nevytvořila na objekt nový, byla by část kódu aplikace nedosažitelná a aplikace by tak v lepším případě „ztratila“ určitou funkcionalitu. V tom horším by nemusela být schopna dalšího provozu vůbec.

Pátý krok, přenos stavu aplikace do nového kódu, také nebývá u všech aktualizací. Jakmile zavedeme do aplikace nový kód, vytvoříme z něj objekty a provedeme přesměrování referencí, budeme mít po určitou dobu v paměti dvě verze aplikace. Starou verzi a novou verzi. Abychom zachovali konzistenci aplikace, je zapotřebí přenést stav ze staré verze aplikace do nové verze. V případě jazyka Java určuje stav aplikace obsah třídních a instančních proměnných. Znamená to tedy že je potřeba zkopírovat obsah těchto proměnných ze staré verze do nové verze aplikace. Pokud se při aktualizaci zásadně změní i struktura tříd, které obsahují proměnné uchovávající stav aplikace tak, že nelze provést prosté zkopírování obsahu proměnných staré verze do nové, je potřeba provést transformaci stavu. Jednoduchý přenos stavu aplikace, nevyžadující transformaci lze provést zcela automaticky. Pokud ale aplikace vyžaduje transformaci stavu z důvodů popsaných výše, je požadován zásah tvůrce aplikace, neboť jen on ví, jak se změnila struktura v nové verzi aplikace. Nelze totiž obecně jednoznačně rozhodnout, která proměnná do které má být přenesena. Jedno z možností řešení tohoto problému je dodání informace o mapování staré struktury aplikace na novou, tvůrcem aplikace. Teprve podle mapovacího schématu je možné automaticky provést transformaci stavu.

Šestý krok, pokračování běhu nové verze aplikace, rovněž nemusí být u všech aktualizací. Jedná se o rozběhnutí nové verze aplikace tak, aby pokračovala od místa, ve kterém byla předchozí verze pozastavena.

Sedmý krok, odstranění staré verze aplikace, je volitelný a nezávisí na něm provoz aplikace samotné. Pokud se provede, bude ušetřeno místo v paměti. Z hlediska Javy to znamená odstranění všech referencí na objekty staré verze aplikace. O samotné odstranění objektů z paměti se postará v případě potřeby Garbage collector.

2.1.4 Rozdělení aktualizací podle činností

Z hlediska prováděných činností si můžeme aktualizace rozdělit do dvou základních skupin. Manuální a automatické. Kdy můžeme mluvit o automatické aktualizaci ? Výše popsané základní kroky aktualizace si můžeme pro tyto potřeby objasnění přerozdělit. Bude se tedy jednat o tyto kroky:

- Kontrola nové verze
- Stažení nové verze
- Instalace nové verze
- Spuštění nové verze

A nyní jedno zamyšlení. Kde je hranice mezi manuálním a automatickým ? Pokud bude muset uživatel aplikace provést všechny kroky sám, je jasné že budeme mluvit o manuální aktualizaci. Pokud se všechny kroky provedou „samy“, bez nutnosti zásahu uživatele nebo kterého koli člověka, je jasné že jde o plně automatickou aktualizaci. Kromě těchto dvou možností, mohou nastat ještě další tři. Automaticky se provede pouze kontrola nové verze a zbývající kroky budou provedeny manuálně. Automaticky se provede kontrola a stažení nové verze. A poslední možnost. Automaticky se provede kontrola, stažení a také instalace nové verze.

Z výše vyjmenovaných čtyř kroků je ten poslední pro člověka nejjednodušší. Co se jednoduché implementace aktualizace týče, pak je pro ní poslední krok naopak nejsložitější. Tudíž v této práci budu považovat za automatickou aktualizaci takovou, která zajistí alespoň první tři kroky a na uživateli ponechá pouze znovu spuštění aplikace.

2.2 Podpora aktualizací v JVM

2.2.1 Spouštění Java aplikací

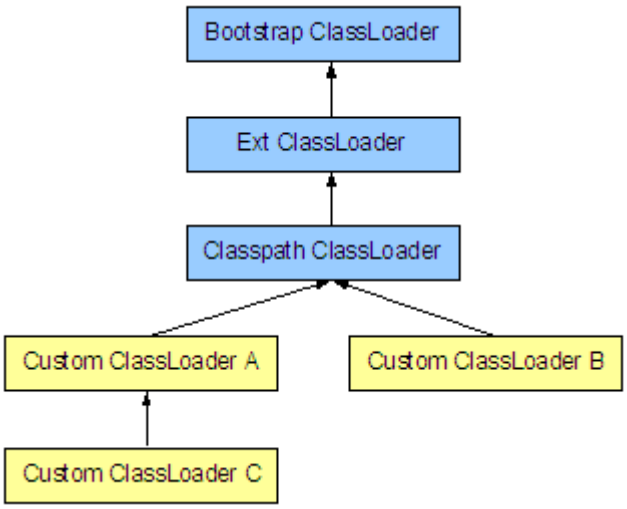
Běh každé aplikace napsané v jazyce Java je umožněn virtuálním strojem tohoto jazyka nazvaným „Java Virtual Machine“ (JVM). Tento virtuální stroj běží na konkrétní platformě (Windows, Linux, MacOS, atd) a odstiňuje Java aplikaci od této platformy. Java aplikace tak není závislá na platformě počítače, na kterém běží. Pokaždé, když se má spustit nějaká Java aplikace, je automaticky také spuštěn virtuální stroj jazyka Java. Jinak by aplikace nebyla schopna běhu. Tento proces probíhá ve stručnosti takto. Po spuštění, najde JVM hlavní třídu aplikace, která má být spuštěna, zavede ji do paměti a zavolá její hlavní metodu. Tímto je daná aplikace spuštěna. Každou další třídu, která se v aplikaci používá, zavádí JVM až ve chvíli, kdy má být táto třída použita.

2.2.2 Zavádění tříd Java aplikací

Zavádění tříd do paměti probíhá pomoci mechanismu „class loaderů“. Class loader je objekt (instance třídy ClassLoader nebo jejích potomků), který umožňuje zavedení binární reprezentace určité třídy do paměti a vytvoření instance třídy Class zavedené třídy. Všechny třídy class loaderů jsou napsány v jazyce Java, kromě jednoho, který je přímou součástí JVM a nazývá se Bootstrap class loader (systémový class loader).

V JVM existuje hierarchie class loaderů, která používá delegační model pro zavádění tříd do paměti. Na vrcholu hierarchie class loaderů stojí Bootstrap class loader, který je rodičem všem ostatním class loaderům. Rodič je v této hierarchii brán ve smyslu delegování (zastoupení), ne ve smyslu dědičnosti. Každý class loader kromě Bootstrap class loaderu, má tedy svého rodiče. Proces zavádění tříd se

uskutečňuje nejdříve pověřením svého rodičovského class loaderu o zavedení určité třídy, pokud rodičovský class loader neuspěje při zavádění určité třídy, teprve pak se o to postará class loader sám.



Obr. 2. Hierarchie class loaderů
Zdroj [2]

Úkoly jednotlivých class loaderů:

- Bootstrap class loader – zavádí základní třídy z Java API
- Ext class loader – zavádí třídy z extension adresáře Javy
- Classpath class loader – zavádí třídy uvedené v classpath
- Custom class loader – vlastní implementace class loaderu, zavádí třídy odkud koli

Standardně tedy v rámci delegování zavádí třídy buďto Bootstrap class loader nebo Classpath class loader. Měně častěji Ext class loader. Jak již bylo řečeno, dochází k zavádění tříd teprve v čase jejich prvního použití. Každá třída je zaváděna tím class loaderem, který zavedl její „volající“ třídu. Volající třída je ta, která používá nějakou jinou třídu pro svojí činnost a udržuje si na ní odkaz. Například máme-li třídu „A“, která byla zavedena Classpath class loaderem a táto třída v určitém kroku nějaké operace vytváří první instanci třídy „B“, je nejprve třída „B“ zavedena do paměti rovněž Classpath class loaderem.

Po prvním zavedení určité třídy do paměti již není možné tuto třídu znovu zavést tou samou instanci class loaderu. JVM zajišťuje, aby nedošlo k vytvoření duplicit tříd. Je ale možné zavést jednu třídu několika různými instancemi určitého classloaderu. Pokud je třída jednou zavedena do paměti, nelze ji z paměti jen tak „vymazat“. Pokud bude existovat reference na instanci class loaderu, který ji do paměti zavedl, nelze tuto třídu z paměti nijak odstranit. Pokud nastane situace, při které již danou třídu nebudeme používat a požadujeme její odstranění z paměti, je nutné nejdříve odstranit všechny reference na class loader, který ji do paměti zavedl a po té odstranit také všechny reference na instance této třídy. Ani v tomto případě nelze třídu z paměti manuálně odstranit. Třída bude odstraněna automaticky až při volání garbage collectoru.

Více informací o class loaderech se lze dozvědět ze zdrojů [1] a [2].

2.2.3 JVM a aktualizace

Jelikož je Java objektově orientovaný jazyk, který je založený na třídách, je jednotkou programu třída. Nejmenší jednotkou v případě aktualizace bude tedy třída. Z tohoto pohledu si lze pod pojmem aktualizace představit nahrazování jedné či více stávajících tříd za nové nebo jiné třídy. Virtuální stroj jazyka Java však přímo nepodporuje nahrazování tříd, tak jak bychom potřebovali. Jak již bylo dříve zmíněno, jednou zavedená třída zůstává v paměti, dokud existuje odkaz na třídu samotnou a na class loader, který ji zavedl. Jelikož většina aplikací nemá vlastní class loader pro zavádění svých tříd, je celá takováto aplikace zavedena Bootstrap nebo ClassPath class loaderem. Tyto class loadery nelze z JVM odstranit. Z toho vyplývá, že z takto koncipované aplikace nelze jednou zavedené třídy vůbec odstranit. Naskýtají se tedy dva základní odlišné přístupy jak docílit možností aktualizace Java aplikací.

První cestou je modifikace JVM tak, aby podporoval znovu zavádění již jednou zavedených tříd. Takto upravenému JVM by pak stačilo změnit obsah souboru třídy uložené někde na disku a zavolat například metodu „refresh“. Tento přístup by měl jednu velkou výhodu. Jaká koli aplikace by bez větších změn ve vlastním kódu byla schopna aktualizace. Ovšem na druhou stranu by zde byla i jedna velká nevýhoda. Podmínkou by totiž byla nutnost používat modifikovanou verzi JVM.

Druhou cestou je pak rozšíření aplikace o sadu podpůrných knihoven a upravených tříd, které zajistí aktualizační procesy nebo „vsazení“ celé aplikace do nějakého rámce vyvinutého pro tyto účely. Tento způsob možná vyžaduje individuálnější přístup a klade větší nároky na implementaci, ale je zcela nezávislý na JVM.

2.3 Možná řešení aktualizací v jazyce Java

2.3.1 Java aplikace

Upřesněme si nejdříve pojem „Java aplikace“. Aplikace napsaná v jazyce Java se mimo jiné skládá z jedné nebo z několika tříd. Tyto třídy jsou obsaženy v jenom nebo více souborech uložených například na pevném disku. Soubory jsou pojmenovány různě ale jejich přípona je vždy „class“. Java aplikace může být tedy tvořena jediným souborem s příponou „class“, ale také několika takovými soubory. Kromě tohoto typu souborů může aplikace obsahovat spoustu dalších typů.

Všechny soubory, tvořící dohromady jednu aplikaci, mohou být umístěny v nějakém adresáři nebo mohou být zabaleny do jednoho či více tak zvaných JAR souborů. JAR (Java archive) je kompresní souborový formát založený na kompresi ZIP. Pomocí těchto souborů lze v jazyce Java například tvořit knihovny tříd. Výhodnou vlastností JAR souborů je také zmenšení výsledné velikosti aplikace použitím komprese.

V této práci bude tedy pojem Java aplikace znamenat jeden nebo více souborů JAR obsahující všechny potřebné Java třídy. Případné další soubory jako obrázky, xml soubory pro konfiguraci, atd, mohou být umístěné také v těchto JAR souborech nebo v nějaké struktuře adresářů. Adresář, ve kterém je celá aplikace obsažena, budeme nazývat „Kořenový adresář aplikace“.

2.3.2 Nahrazování původních souborů novými

Při spouštění Java aplikace, jak již bylo řečeno, hledá JVM hlavní třídu, aby mohl zavolat metodu „main“ a tím spustit aplikaci. Hlavní třída aplikace je jako všechny ostatní třídy zabalena v JAR souboru. JVM si tedy načte JAR soubor, prohledá jeho strukturu a učiní všechny potřebné kroky vedoucí ke spuštění aplikace. Po načtení takového JAR souboru si JVM tento soubor „uzamkne“ proti smazání.

Při aktualizaci tedy nelze použít postup, kdy nejdříve všechny stávající soubory aplikace smažeme a poté na jejich místo zkopírujeme soubory nové. Tedy alespoň ne u JAR souborů. Souborový systém to z důvodu uzamčení těchto souboru nedovolí. Co ale udělat lze, je přepsat původní JAR soubory novými verzemi těchto souborů. Na první pohled by se zdálo, že tady by proces aktualizace mohl skončit. Tak tomu ale není. Přepsání obsahu původních JAR souborů novým obsahem způsobí totiž dvě základní věci. Za prvé, nový obsah JAR souboru nebude mít vliv na třídy, které již byly zavedeny do paměti. Tím pádem se aktualizace nijak neprojeví, dokud se aplikace nerestartuje. Za druhé, při pokusu načíst binární reprezentaci třídy, která ještě nebyla zavedena do paměti, pozná JVM že soubor JAR byl změněn a při načítáni selže. Selhání je doprovázeno vyhozením výjimky o nenalezení požadované třídy. Navíc, pokud bychom do aplikace přidali další, nový soubor JAR, virtuální stroj by o něm nevěděl a ten by proto byl nedosažitelný.

2.3.3 Jednoduchý způsob implementace vyžadující restart

Na obrázku číslo 3 je ukázka implementace hlavní metody velmi jednoduché aplikace. Táto metoda pouze vytvoří instanci aplikace a spustí jí.

```
/*
 * Hlavní metoda, která spouští aplikaci.
 * Standardní, jednoduchá implementace
 * neobsahující aktualizační procesy.
 */
public static void main(String[] args) {
  // Vytvoření instance aplikace
  Aplikace aplikace = new Aplikace();

  // Spuštění vlastní aplikace
  aplikace.start();

  // Po ukončení aplikace se pokračuje zde,
  // ale jelikož tady nic není, aplikace končí.
}
```

Obr. 3. Standardní implementace metody „main“

Nejjednodušší způsob implementace aktualizací do již existující aplikace vyžadující však restart je možné provést takto:

Najdeme přesné místo, odkud se spouští aplikace. Ve většině případů jde o metodu „main“ umístěnou v hlavní třídě. Do této metody je potřeba umístit volání té části kódu aktualizace, která má za úkol ověření dostupnosti nové verze aplikace na serveru. Dejme tomu, že tuto část kódu bude

zastupovat metoda s názvem „zkontrolujDostupnost()“ a její návratová hodnota bude typu boolean. Ta bude nabývat hodnoty „false“ v případě že nebude existovat nová verze aplikace a „true“ v případě opačném. Volání této metody umístíme někde na začátek metody „main“, před první volání vlastního kódu aplikace. Hned za tuto metodu umístíme rozhodovací podmínku, která rozdělí chod aplikace v tomto místě do dvou různých větví. Pokud je návratová hodnota metody „zkontrolujDostupnost()“ „false“, tedy nová verze aplikace není dostupná, bude se pokračovat větví číslo „1“, která povede k volání vlastního kódu aplikace a aplikace tak zahájí svojí činnost. Pokud však bude návratovou hodnotou „true“, což znamená že nová verze aplikace dostupná je, bude se pokračovat větví číslo „2“, která zajistí stažení nových souborů a provedení aktualizačních procesů. V tomto případě bude provedení aktualizačních procesů spočívat v prostém nahrazení (přepsání) stávajících souborů aplikace novými (změnovými) soubory, případně přidáním některých dalších nových souborů. Po této činnosti je nutný restart aplikace. Takže posledním krokem této větve může být zobrazení dialogového okna, ve kterém informujeme uživatele aplikace o provedené aktualizace a nutném restartu aplikace. Po stisknutí jediného možného tlačítka „OK“ se aplikace ukončí. Opětovné spuštění pak musí zajistit uživatel.

```
/*
 * Hlavní metoda, která spouští aplikaci.
 * Použitá implementace jednoduché aktualizace
 * vyžadující restart.
 */
public static void main(String[] args) {
  // Proměnná pro rozhodovací podmínku
  private boolean jeAktualizace = false;

  // Vytvoření instance aplikace
  // a instance aktualizace
  Aplikace aplikace = new Aplikace();
  Aktualizace aktualizace = new Aktualizace();

  // Kontrola dostupnosti nové verze aplikace
  jeAktualizace = aktualizace.zkontrolujPritomnostNoveVerze();
  // Rozhodovací pomínka
  if (jeAktualizace) { // Pokud je nalezena nová verze aplikace
    // Provedení aktualizačních procesů
    aktualizace.stahniNoveSoubory();
    aktualizace.aktualizujAplikaci();
    aktualizace.zobrazVyzvuNaRestart();
    // Zde se aplikace ukončí, aby se projevily změny po aktualizaci.
  } else { // Pokud nová verze aplikace neexistuje
    // Spuštění vlastní aplikace
    aplikace.start();
    // Po ukončení aplikace se pokračuje zde,
    // ale jelikož tady nic není, aplikace končí.
  }
}
```

Obr. 4. Upravená implementace metody „main“ – vyžaduje restart

Obrázek číslo 4 ukazuje jak lze provést jednoduchou implementaci aktualizace popsanou výše. Kód není úplný, jsou zde jen ukázána místa, kde se co umísťuje.

2.3.4 Jednoduchý způsob implementace nevyžadující restart

Jednoduchou úpravou předchozího způsobu implementace aktualizací lze dosáhnout toho, že nebude vyžadován restart aplikace. Cenou za toto zvýhodnění je však drobná nevýhoda. Aktualizace aplikace se v případě dostupnosti nové verze provede až na konci aplikace. Restart tedy není nutný, protože po provedení aktualizačních procesů následuje automaticky ukončení aplikace. Nevýhoda může být v tom, že aplikace po spuštění pracuje ve stávající verzi, jelikož je aktualizace posunuta až na konec a teprve až při dalším spuštění se projeví její aktualizace. Zda to bude považováno za nevýhodu záleží na konkrétní aplikaci a konkrétních požadavcích.

Úprava je tedy následující:
Do rozhodovací podmínky se přidá třetí větev, která v případě že je dostupná nová verze aplikace, odloží aktualizaci na konec aplikace. To znamená, že v tomto případě by se po spuštění aplikace opět provedla kontrola na dostupnost nové verze aplikace, opět se v podmínce objeví návratová hodnota „true“, která vybere větev číslo „2“. V tomto místě však nebudeme ihned provádět aktualizační procesy, ale přidáme druhou rozhodovací podmínku. Zde se bude rozhodovat, jestli se má aktualizace provést ihned nebo odložit na konec aplikace. V případě, že se má aktualizace provést ihned, zahájí se aktualizační procesy stejným způsobem, jak je popsáno výše. V druhém případě, kdy se má aktualizace odložit, se může ale nemusí provést pouze stažení nové verze aplikace někam do dočasného adresáře, ale hlavně je nutné zapamatovat si, že provedení aktualizace samotné se uskuteční až při ukončování aplikace. Po té se spustí vlastní kód aplikace. Aktualizační procesy se pak vloží do místa, kde se volá ukončení aplikace nebo prostě za poslední příkaz v aplikaci. Aby se aktualizační procesy neprováděly vždy, i když není dostupná nová verze aplikace, je třeba ještě před jejich provedením ověřit, jestli je potřeba je provádět. Zapamatování si odložení aktualizace lze provést zavedením proměnné typu boolean s názvem například „jeAktualizace“. Ověřování, zda se má při ukončování aplikace provést aktualizace by pak probíhalo dotazem právě na tuto proměnnou.

Druhá rozhodovací podmínka zde není nutná. Aplikaci můžeme nastavit tak, aby se aktualizace prováděla vždy při ukončování. Zavedením této podmínky však dostáváme možnost, kdykoli změnit umístění aktualizace (na začátku nebo na konci). Rozhodování, který způsob aktualizace zvolit se pak může provádět pomoci dialogového okna, kdy po spuštění aplikace vyzveme uživatele aby zvolil jednu z těchto dvou možností. Další možnost je umístit rozhodování do konfiguračního souboru.

Pokud umístíme rozhodování do konfiguračního souboru a nastavíme jej tak aby se aktualizace prováděly na konci aplikace, nebude v průběhu aplikace od spuštění až po ukončení vyžadována uživatelská interakce. To lze s výhodou využít u aplikací, které se spouštějí automaticky podle nějakého nastaveného plánování bez zásahu uživatele.

Obrázek číslo 5 ukazuje druhý způsob jednoduché implementace aktualizace do aplikace. Ani zde není kód kompletní, ale je pouze naznačeno, kde se co umísťuje. Tento způsob je mírně složitější než předchozí, ale umožňuje přepínání mezi oběma způsoby aktualizací. Přepínání se uskutečňuje pomoci proměnné „odlozitAktualizaci“. Pokud má táto proměnná hodnotu „true“, bude se aktualizace odkládat na konec aplikace a nebude vyžadován restart.

```
/*
 * Hlavní metoda, která spouští aplikaci.
 * Použitá implementace jednoduché aktualizace vyžadující restart.
 */
public static void main(String[] args) {
  // Proměnná pro rozhodovací podmínku
  private boolean jeAktualizace = false;
  // Proměnná pro přepínání mezi umístěním aktualizace
  // na začátek nebo na konec aplikace
  private boolean odlozitAktualizaci = true;

  // Vytvoření instance aplikace a instance aktualizace
  Aplikace aplikace = new Aplikace();
  Aktualizace aktualizace = new Aktualizace();

  // Kontrola dostupnosti nové verze aplikace
  jeAktualizace = aktualizace.zkontrolujPritomnostNoveVerze();
  // První rozhodovací pomínka
  if (jeAktualizace) { // Pokud je nalezena nová verze aplikace
    // Stáhnutí nových souborů do dočasného adresáře
    aktualizace.stahniNoveSoubory();
    // Druhá rozhodovací podmínka
    if (!odlozitAktualizaci) { // Aktualizace se mají vykonat ihned
      // Provedení aktualizačních procesů
      aktualizace.aktualizujAplikaci();
      aktualizace.zobrazVyzvuNaRestart();
      // Zde se aplikace ukončí, aby se projevily změny po aktualizaci
      System.exit(0);
    }
  }
  // Spuštění vlastní aplikace
  aplikace.start();
  // Po ukončení aplikace se pokračuje zde
  if (odlozitAktualizaci) { // Je třeba vykonat odloženou aktualizaci
    // Provedení aktualizačních procesů
    aktualizace.aktualizujAplikaci();
  }
}
```

Obr. 5. Upravená implementace metody „main“ – nevyžaduje restart

2.3.5 Implementace opakovaných aktualizací nevyžadujících restart

Předchozí dva způsoby jsou jednoduchou implementaci aktualizací, které však vyžadují přerušení aplikace aby se projevily změny. Táto řešení se však nedají použít tam, kde je vyžadován trvalý běh aplikace bez přerušení. Kdyby se totiž posunula aktualizace na konec takovéto aplikace, nebylo by předem vůbec známo, kdy se vlastně taková aktualizace provede, jelikož aplikace může běžet nepřetržitě třeba půl roku. Pokud bychom provedli aktualizaci na začátku, dalo by se to udělat jen jednou za dlouhé období. V takových to případech je potřeba použít složitější způsob implementace,

který umožní aktualizace provádět opakovaně tak, aby se změny projevily ihned po ukončení aktualizačních procesů bez jediného přerušení aplikace.

Přerušení aplikace (restart) je až doposud nutné z důvodu projevení změn po aktualizaci. Změny vyžadují restart aplikace například proto, že třídu jednou zavedenou do paměti již nelze za provozu zavést podruhé. Tím pádem zůstává v paměti pořád stará verze třídy i když je již dostupná nová verze. Prvním úkolem je tedy zajistit, aby se dala třída za provozu zavést vícekrát do paměti. Tím umožníme nahrazení staré verze třídy verzi novou. Druhým úkolem je zajistit, aby virtuálnímu stroji nevadilo, že mu byl za provozu změněn obsah JAR souboru. Tím umožníme načítat třídy ze změněného JAR souboru aniž by došlo k vyhození výjimky a zastavení aplikace. Třetím úkolem je pak zajistit, aby v případě že se po aktualizaci objeví v aplikaci nový JAR soubor, byl tento soubor dosažitelný. Tedy aby se o něm „dozvěděl“ virtuální stroj a mohl tak využívat nové třídy, a knihovny tříd. Pokud tedy splníme tyto tři podmínky, nebude nutné aplikaci po aktualizaci restartovat aby se projevily změny v aplikaci.

Zajištění opakovaného zavádění třídy do paměti:

Protože nelze jednu a tutéž třídu zavést do paměti vícekrát jednou instancí class loaderu, je potřeba zajistit před dalším zaváděním té samé třídy do paměti odstranění reference na již použitou instanci class loaderu, který ji zavedl a vytvořit instanci novou. Abychom toho byli schopni, musíme používat jiný než Bootstrap, Ext nebo ClassPath class loader. To nás vede k tomu, abychom pro zavádění tříd naší aplikace používali custom class loader. Viz obrázek s přehledem class loaderů výše. Chceme li však, aby třídy skutečně zaváděl náš custom class loader a ne jiný, musíme zajistit, aby JAR soubory a třídy obsažené v těchto souborech našel pouze custom class loader a žádný jiný. Pokud naše třídy bude schopný najít i ClassPath class loader nebo jiný systémový class loader, zavede je ten, z důvodu delegačního modelu zavádění tříd v Javě.

Abychom tedy splnili výše uvedené podmínky, musíme udělat dvě věci. Za prvé, předefinovat metodu „findClass“ našeho custom class loaderu tak, aby byla schopna najít všechny naše JAR soubory a v těch všechny potřebné třídy. Implementace této metody závisí na konkrétní aplikaci. Za druhé, je potřeba „ukrýt“ všechny třídy aplikace před ClassPath class loaderem, aby je nemohl zavést on. Hlavní třídu, obsahující metodu „main“, která spouští aplikaci však nelze zavést jinak, než jedním ze systémových class loaderů. Z tohoto důvodu bude nutné upravit spouštění aplikace. V tomto případě již nelze spouštět aplikaci přímo, ale bude potřeba ji umístit do takzvaného „rámce“. Přímo se bude spouštět tento rámec, tudíž systémový class loader zavede do paměti pouze jeho třídy. Rámec po svém spuštění zajistí vytvoření instance custom class loaderu a hned na to pomocí tohoto class loaderu zavede třídy samotné aplikace do paměti a spustí aplikaci zavoláním její metody „main“. Takto vlastně zamezíme zavedení jakékoli třídy aplikace jedním ze systémových class loaderů. Manifest, který obsahuje atribut Class-path, ve kterém jsou uvedeny cesty k dalším použitým JAR souborům aplikace je potřeba změnit. Změna je jednoduchá. Odstraníme tento atribut se všemi cestami z manifestu pryč, čímž „ukryjeme“ všechny JAR soubory aplikace, na které se v tomto atributu ukazuje, před ClassPath class loaderem.

Umožnění změn v JAR souboru za provozu

Z předchozích odstavců vyplývá, že je nutné použít pro zavádění tříd aplikace vlastní class loader. Použití vlastního class loaderu řeší i problém změny obsahu JAR souboru za běhu aplikace. V našem class loaderu vytvoříme metody, které zajistí tyto činností:

- Nalezení všech JAR souborů aplikace – metoda, zajišťující tuto činnost, postupně prohledá všechny adresáře aplikace a vytvoří seznam všech JAR souborů, které se v ní nalézají.
- Vytvoření seznamu všech Java tříd aplikace – metoda, zajišťující tuto činnost, projde postupně všechny JAR soubory nalezené předchozí metodou a vyhledá v těchto JAR souborech všechny

Java třídy, které jsou v nich obsažené. Vytvoří seznam, do kterého umístí jména všech nalezených tříd i s informací, ve kterém JAR souboru se daná třída nalézá. K tomuto účelu bude vhodné použít kolekci zvanou HashMap, kde jména tříd budou klíčem a jména JAR souborů, hodnotou.

- Nalezení konkrétní třídy, kterou je potřeba zavést – metoda zajišťující tuto činnost je výše zmíněná metoda „findClass“. Tuto upravíme tak, aby prohledávala náš seznam tříd.
- Načtení binární reprezentace dané třídy – metoda, zajišťující tuto činnost, obdrží jméno konkrétní třídy, kterou je potřeba zavést do paměti, podle tohoto jména si vyhledá v předem vytvořeném seznamu tříd informace o JAR souboru, ve kterém je obsažena daná třída a z tohoto JAR souboru načte její binární reprezentaci do pole bajtů.
- Nalezení hlavní třídy aplikace – metoda, která zajišťuje tuto činnost, prohledá všechny nalezené JAR soubory na přítomnost manifestu a z manifestu zjistí název třídy, která je určena jako hlavní.
- Zavedení aplikace do pamětí – metoda, která zajišťuje tuto činnost, zavede nalezenou hlavní třídu aplikace do paměti.
- Znovu vytvoření seznamu všech tříd aplikace – metoda, zajišťující tuto činnost, vymaže aktuální seznam všech tříd aplikace a postupným voláním metod předchozích činností vytvoří tento seznam znovu.

Při spuštění rámce se nejprve vytvoří instance našeho vlastního class loaderu. Na této instanci se potom postupně zavolají metody, které provedou výše zmíněné činnosti. To zajistí vytvoření seznamu všech tříd aplikace a umožní této instanci class loaderu načítat jejich binární reprezentaci do paměti. Po té rámec najde hlavní třídu aplikace, zavede ji do pamětí pomoci custom class loaderu a zavolá její metodu „main“. Tímto bude aplikace spuštěna. Každá další třída kromě té hlavní, kterou bude aplikace potřebovat, bude zavedena rovněž custom class loaderem, jelikož to zajišťuje virtuální stroj automaticky. Viz podkapitola 2.2.2 Zavádění tříd Java aplikací.

Zajištěni dosažitelnosti nově přidaných JAR souborů

Po každém provedení aktualizace je potřeba zavolat metodu aktuální instance custom class loaderu, která má na starosti znovu vytvoření seznamu všech tříd aplikace. Při vytváření tohoto seznamu se mimo jiné znovu vyhledávají všechny JAR soubory v celé aplikaci. To zajistí, že pokud přibyl v aplikaci po aktualizaci nový JAR soubor, bude zahrnut do seznamu a tím pádem také třídy, které obsahuje. Takto bude nový JAR soubor „viditelný“ pro virtuální stroj aniž by byl nutný restart aplikace.

Před provedením aktualizace takto upravené implementace je nutné nejprve zajistit bezpečný stav aplikace. Co je to bezpečný stav aplikace je popsáno na začátku této kapitoly v podkapitole 2.1.3 Základní kroky aktualizace. Teprve pak, lze provést aktualizaci samotnou. Při provádění aktualizace nejprve zkopírujeme všechny nové soubory do aplikace. Tímto v aplikaci přepíšeme některé stávající soubory a některé zde vzniknou jako zcela nové. Po té vytvoříme novou instanci našeho custom class loaderu a zavoláme postupně jeho metody, které připraví tento class loader na zavádění tříd. Nyní postupně vytváříme nové objekty podle těch, které jsou obsaženy v aktuální struktuře aplikace. Zároveň tyto objekty propojujeme tak aby vytvořily stejnou strukturu jako je ta stávající. Jakmile je nová struktura dokončena, odstraníme všechny reference na starou instanci custom class loaderu a po té také všechny reference na staré objekty. Tím zajistíme, že při spuštění Garbage collectoru budou staré objekty odstraněny z paměti. V této fázi ukončíme provádění aktualizací a předáme řízení opět aplikaci. Aplikace bude pokračovat od místa, kde jsme ji přerušili.

Jak může vypadat custom class loader je ukázáno v příloze číslo 2.

2.3.6 Implementace aktualizací nevyžadující restart s přenosem stavu

Předchozí způsob implementace aktualizace přináší podstatné vylepšení oproti prvním dvěma způsobům. Jeho použitím, již není potřeba aplikaci restartovat po provedení aktualizace, což umožňuje dlouhodobý nepřerušovaný běh aplikace. Má však jedno omezení, které jej nedovoluje použít ve všech aplikacích. Neumožňuje přenos stavu staré verze aplikace do verze nové. Táto problematika je popsána v podkapitole 2.1.2, kde jsme si rozdělili aplikace podle pamatování si stavu nebo dat. Předchozí způsob implementace lze tedy použít jen v těch aplikacích, které si nemusí nic pamatovat. Pokud máme aplikaci, kde jsou obsažena nějaká data, které je nutno uchovávat po celou dobu běhu aplikace nebo si aplikace musí pamatovat nějaké své vnitřní stavy, musíme předchozí verzi implementace upravit tak, aby nedošlo ke ztrátě dat nebo informací o stavech.

Vše, co si Java aplikace musí „pamatovat“ během svého provozu je uloženo v třídních nebo instančních proměnných. Celkový stav aplikace z pohledu aktualizace je tedy dán aktuálním obsahem těchto proměnných. Předchozí implementaci tedy musíme rozšířit o možnost přenosu obsahu proměnných staré verze aplikace do proměnných nové verze. Vytvoříme tedy mechanismus, který toto bude zajišťovat a uvedeme jej v činnost, jakmile bude vytvořena nová struktura aplikace. Do této fáze je postup aktualizace stejný jako v předchozí implementaci. Teprve po přenosu celkového stavu aplikace budeme odstraňovat reference na staré objekty.

Vytvoření mechanismu, který má za úkol přenést stav staré verze aplikace do nové verze nemusí být zrovna jednoduchý. Po aktualizaci totiž mohou nastat dva základní případy pro každou třídu, která obsahuje proměnné uchovávající data nebo informace. V prvním případě se jména a počet proměnných nezmění a přenos stavu mezi třídami není složitý. S pomocí reflexe se postupně naleznou a zkopírují všechny proměnné každé takové třídy, případně její instance. Pokud se však změní jména proměnných a/nebo jejich počet, je situace obtížnější a vyžaduje poskytnutí dodatečných informací od tvůrce aktualizace. Tyto informace říkají, jak se má provést přenos stavu mezi třídami v takovém případě. Pomocné informace obsahují buď mapování názvu proměnných mezi kterými má proběhnout přenos obsahu nebo transformační logiku, případně obojí. Transformační logika pak může zajišťovat i složitější převody a postupy při přenosu stavu mezi třídami případně jejich instancemi.

Podrobné informace o technikách zajišťující konzistenci aplikace se lze dočíst v [3].

3 Podpora automatických aktualizací

Táto kapitola poskytuje základní přehled některých nástrojů podporujících implementaci automatických aktualizací.

3.1 Přehled projektů

3.1.1 Internetový zdroj „http://sourceforge.net“

Na stránkách „sourceforge.net“ je spousta menších projektů, které se zabývají tématem aktualizací Java aplikací. Většina z nich má však rozepsaných pouze „pár“ stránek zdrojového kódu. Některé jsou v začátcích a je vidět, že zdrojové kódy přibývají, jiné stagnují třeba i několik měsíců. U pár projektů jde vidět, že se jím vývojáři opravdu věnují a existují už i funkční verze hotového frameworku či knihovny. S dokumentací či popisem, jak daná knihovna či framework funguje, je to velmi slabé. Uvedu zde proto několik odkazů na jednotlivé projekty a ty, u kterých je dostupná nějaká dokumentace nebo popis popíšu více do hloubky.

Seznam projektů, zabývajících se aktualizacemi Java aplikací.

- Java Update Tool (JUT) - https://sourceforge.net/projects/jut
- Java Update Framework (JUF) - https://sourceforge.net/projects/juf
  - Rámec, který zjednodušuje implementaci automatických aktualizací.
- Java Automatic Update (JUP) API - https://sourceforge.net/projects/jau-api/
  - Podporuje vývoj desktopových Java aplikací
  - Autorizovaný přístup, konfigurace založena na XML souborech
- Jar File Updator - https://sourceforge.net/projects/jarupdator
  - Java knihovna pro aktualizaci JAR souborů
- Update 4 Java - https://sourceforge.net/projects/update4java
  - Při startu aplikace ověří verze a upozorní uživatele
- EasyUpdate - https://sourceforge.net/projects/myupdater
  - Java knihovna umožňující aplikacím kontrolu nové verze, stažení a instalaci.
  - Používá elektronický podpis souborů, který chrání aplikaci před podvrhem.
- Java Application Update Utility Software (JAUUS) - http://sourceforge.net/projects/jauus
- stableUpdate - http://stableupdate.sourceforge.net/index.htm

3.2 JAUUS - Java Application Update Utility Software

3.2.1 Vznik projektu

Tento projekt založil autor na základě potřeby aktualizovat svou vlastní aplikaci. Napsal aplikaci pro nějaké oddělení podpory, která z různých důvodů vyžadovala pravidelně provádět aktualizaci. Z počátku se aktualizace prováděla tak, že autor vytvořil nové soubory, umístil je na předem určené

místo a obeslal všechny jež používali jeho aplikaci e-mailem, ve kterém dotyčné informoval o nové verzi aplikace. Ne všichni však jeho e-maily četli nebo si případně nevěděli rady s aktualizaci. Tohle řešení tedy nebylo moc spolehlivé a hlavně efektivní. Proto vznikl tento projekt.

Projekt má i své internetové stránky.
Adresa domovské stránky je „http://jauus.sourceforge.net/index.html“.
Adresa stránky pro vývojáře je „http://sourceforge.net/projects/jauus“.

3.2.2 Základní popis funkce

Celý systém pracuje na principu klient / server. Server je centrálním místem, kde jsou umístěny vždy nejnovější verze jednotlivých aplikací. Server totiž dokáže obsluhovat více než jednu aplikaci.

Klient v tomto systému funguje jako samostatná část pracující nad danou aplikací. Nespouští se tedy přímo aplikace, ale klient. Ten se po spuštění připojí k serveru a ověří jestli je dostupná nová verze aplikace. Pokud ano provede stažení potřebných souborů, aktualizuje aplikaci a po té ji případně spustí. Pokud nová vrze není dostupná, rozhoduje pouze o spuštění aplikace. To, jestli se má aplikace po ukončení spojení se serverem automaticky spustit nebo ne, je dáno nastavením příslušného atributu v konfiguračním souboru aplikace. V tomto souboru je také uveden příkaz, který se má při spuštění aplikace volat a jeho seznam parametrů, se kterými se volá. Klient má také svůj konfigurační soubor. Ten obsahuje informace jednak o serveru, IP adresu, port, na kterém server naslouchá a doménové jméno serveru a také informace o aplikaci. Především jméno aplikace, které je důležité předat serveru po navázání spojení, aby věděl se kterou aplikací se bude pracovat a informaci o tom co dělat v případě že se spojení nepovede navázat. Zda-li se má aplikace automaticky spustit nebo ne.

3.2.3 Princip kontroly nové verze

Táto aktualizační knihovna se na všechny soubory, které aplikace používá, dívá jako na obecné objekty, které jsou si rovny. Nerozlišuje, je-li ten, či jiný soubor obrázek, xml konfigurace, audio soubor nebo binární kód. Je to prostě soubor uložený na pevném disku. Jediné co ji zajímá, je kontrolní součet každého použitého souboru a následně pak kontrolní součet celé aplikace (všech souborů dohromady). Táto vlastnost má velkou výhodu v tom, že takto lze aktualizovat jakoukoli aplikaci napsanou v jakém-koli jazyce. Nezáleží na programovacím jazyku ani na platformě, na které aplikace běží.

Při navázání komunikace se serverem nejprve klient pošle na server žádost o provedení kontroly na přítomnost nové verze spolu se jménem aplikace, o kterou se „stará“. Server pak vypočítá kontrolní součty každého souboru dané aplikace a nakonec celé aplikace. Tyto informace pošle klientovi. Klient udělá to samé u sebe. Po té porovná kontrolní součet celé své aplikace s celkovým kontrolním součtem obdrženým ze serveru. Jsou-li stejné, znamená to že aplikace nacházející se u klienta je stejná jako ta na serveru. Tedy není potřeba provádět aktualizaci. Pokud se celkový součet liší, začne klient porovnávat kontrolní součty jednotlivých souborů. Soubor u kterého je zjištěn rozdíl v součtech je považován za změněný a je stažena ze serveru jeho nová verze. Soubor, který u klienta není nalezen je považován za nový a je rovněž stažen ze serveru. Soubor, který naopak není nalezen na serveru, ale u klienta existuje, je považován za starý a neplatný. Nové a změněné soubory se po stažení nakopírují do aplikace a ty staré neplatné se z aplikace smažou. Server v případě provádění aktualizace rovněž posílá klientovi i konfigurační soubor dané aplikace, který obsahuje informace o tom jak se aplikace spouští a jestli jí má spustit automaticky.

3.2.4 Závěr a možné použití

Kontrola na novou verzi aplikace se provádí pouze na začátku, ještě před spuštěním aplikace. Případná aktualizace pak pouze nakopíruje nové soubory, přepíše stávající a staré soubory smaže. Z těchto důvodů lze tuto knihovnu použít pouze v aplikacích, které nevyžadují trvalý provoz. Aktualizace by se totiž nedala používat opakovaně bez přerušení aplikace. Zde se nedá mluvit o restartu aplikace po aktualizaci, protože knihovna běží samostatně odděleně od aplikace. Jelikož je aktualizace umístěna na začátek, projeví se změny ihned po provedení aktualizace. Velkou výhodou je však možnost použití i pro jiné programovací jazyky než je Java.

3.3 stableUpdate

3.3.1 Základní popis funkce

StableUpdate je knihovna, rozšiřující Java aplikace o možnost automatické aktualizace. Je napsána v jazyce Java jako nezávislý modul. Táto knihovna vyžaduje, aby daná aplikace, která ji hodlá používat, obsahovala podadresář „bin“, ve kterém bude umístěna jak táto knihovna, tak ještě minimálně jeden JAR soubor, ze kterého bude knihovna volána.

Při použití této knihovny může aplikace běžet až do chvíle vlastní aktualizace. Před započetím vlastní aktualizace je aplikace zastavena knihovnou a její virtuální stroj taktéž. Po té se teprve aktualizují soubory aplikace. Virtuální stroj aplikace se zastavuje z důvodu odemknutí JAR souborů, které by jinak nešly aktualizovat. Přesněji by nešlo je pouze mazat v případě potřeby. Knihovna umožňuje zadat několik URL adres serverů obsahujících nenovější verze aplikace, ke kterým se pak připojuje. To zvyšuje dostupnost v případě selhání některého z nich.

Táto knihovna nejen že umožňuje aktualizaci aplikace, která ji implementuje, ale umožňuje i „odvolání“ aktualizace (návrat aplikace k původní verzi) v případě, že se po aktualizaci vyskytnou problémy.

4 Návrh vlastního systému pro automatické aktualizace

4.1 Požadavky na systém

4.1.1 Základní požadavky

- Vytvořit systém automatických aktualizací pro Java aplikace
- Umožnit Java aplikacím automaticky kontrolovat dostupnost nové verze po jejich spuštění.
- Umožnit stahování pouze změněných nebo nových souborů.
- Umožnit kontrolu licenčních klíčů k aktualizaci.
- Vytvořit server pro poskytování nejnovější verze aplikace

4.1.2 Odvozené požadavky

Z předchozích, základních požadavků vyplývají další upřesňující požadavky. Jelikož má systém umožňovat aktualizaci různým Java aplikacím a ne jen jedné konkrétní, musí být napsán obecně a být co nejvíce samostatný. Čím obecnější bude celý systém, tím jednodušší pak bude implementace do konkrétní aplikace. To platí i samostatnosti systému. Samostatnost systému aktualizací bude navíc zvýhodňovat aplikace, které jsou již napsány a musejí být upraveny pro používání dalšího „cizího“ kódu. Má-li systém umožnit stahování jen některých souborů (změněných a nových), nestačí rozlišovat pouze verzi aplikace samotné, ale je potřeba vytvořit funkcionalitu pro rozlišování verzí na úrovní jednotlivých souborů. To znamená, že se bude kontrolovat každý soubor obsažen v aplikaci zvlášť.

Souhrn odvozených požadavků:

- Vytvořit systém aktualizaci obecně
- Vytvořit systém aktualizací jako samostatný
- Umožnit rozlišování verzí na úrovni jednotlivých souborů

4.1.3 Doplňující požadavky

Při vývoji nových verzí software občas dochází k vydání verze obsahující chybnou funkcionalitu určité části aplikace. V takovém případě se řada uživatelů rozhodne vrátit se zpět k předchozí verzi dané aplikace a počkat až se chyby odstraní. Další požadavky na tento systém tedy jsou možnost provádění zálohy aktuální verze před aktualizací a umožnění následné obnovy ze zálohy.

Souhrn doplňujících požadavků:

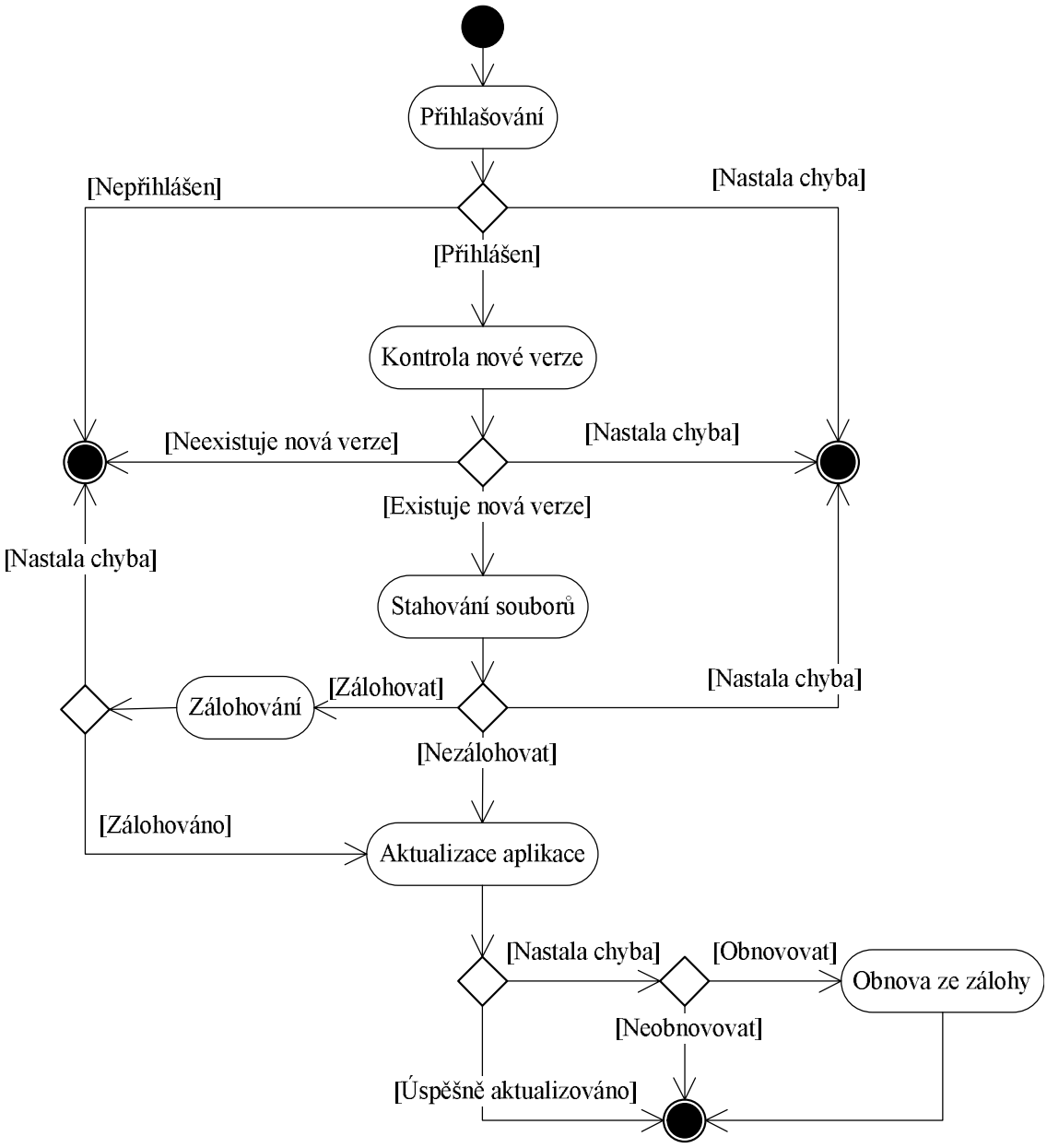
- Možnost zálohovat aktuální verzi aplikace před provedením aktualizace
- Možnost obnovy předchozí verze aplikace ze zálohy

4.2 Analýza systému

4.2.1 Aktivity systému

Proces celého systému lze rozdělit na několik aktivit (kroků). Protože jedním z požadavků je možnost kontrolovat licenční klíče k aktualizaci a omezit tak přístup aplikací na server, bude první aktivita systému přihlášení k serveru. Při této aktivitě se odešle na server licenční klíč, na základě

kterého se rozhodne, zda má aplikace povolen přístup na server nebo ne. Po provedení této aktivity mohou nastat dva základní případy, které zásadně ovlivní proces aktualizace. Pokud přihlášení z kterého koli důvodu selže, nelze provést kontrolu dostupnosti nové verze a proces aktualizace zde končí. V případě úspěšného přihlášení se přejde k další aktivitě. V případě selhání přihlášení je také možno rozlišovat příčiny tohoto selhání a na jejich základě se pokusit o nápravu příčiny a pokus o přihlášení opakovat.



Obr. 6. Diagram aktivit systému automatických aktualizací

Druhou aktivitou v procesu aktualizace bude kontrola dostupnosti nové verze aplikace. Na straně aplikace se vytvoří struktura aktuální verze aplikace, která bude obsahovat všechny názvy souborů

aplikace včetně jejich verzí. Táto struktura se odešle na server s požadavkem na kontrolu verzí. Server vytvoří stejným způsobem strukturu na své straně a porovná ji se strukturou aplikace. Na základě porovnání obou struktur odešle server zpět aplikaci buď oznámení o tom, že verze jsou si rovny a aktualizace není potřeba nebo seznam akcí, které je potřeba provést k dosažení nové verze. Tímto seznamem dá aplikaci najevo potřebu aktualizace. Výsledkem této aktivity mohou být tedy dva případy. Nová verze buďto je dostupná nebo není. Pokud nová verze dostupná není, proces aktualizace končí. V případě dostupnosti nové verze se přechází k další aktivitě.

Třetí aktivitou v procesu aktualizace bude stažení nových nebo změněných souborů ze serveru do dočasné složky aplikace. Aplikace na základě seznamu akcí, které obdržela v předchozí aktivitě, postupně stáhne všechny soubory, které je potřeba.

Čtvrtou aktivitou v procesu aktualizace je provedení zálohy stávající verze aplikace. Pro případ potřeby vrátit se k předchozí verzi aplikace je potřeba provést nejdříve zálohu. Táto aktivita může být volitelná. Záloha spočívá ve zkopírování všech souborů aplikace do složky zálohy.

Pátou aktivitou v procesu aktualizace je vykonání všech aktualizačních akcí. Mezi tyto akce patří zkopírování všech nových a změněných souborů z dočasné složky do aplikace a případné smazání již nepotřebných souborů.

Šestou aktivitou v procesu aktualizace je obnova aplikace na předchozí verzi. V případě, že se aktualizaci nepodaří provést úspěšně nebo je nová verze aplikace nevyhovující je možné volitelně provést návrat k předešlé verzi aplikace.

Při všech aktivitách prováděných v procesu aktualizace může dojít k nějaké chybě, což znamená že tok aktivit nemusí být vždy stejný. Pro lepší pochopení a přehled procesu aktualizace jsou všechny aktivity a přechody mezi nimi znázorněny v diagramu aktivit.

4.2.2 Komunikace se serverem

Komunikaci se serverem budou využívat první tři aktivity. Aktivita přihlašování pošle serveru požadavek na přihlášení a s ním také licenční klíč. V případě platnosti licenčního klíče pošle server status kód 200 OK a žádost o uložení cookie jejímž obsahem bude session id pro pozdější identifikaci klientské aplikace. V případě, že licenční klíč bude neplatný, pošle server status kód 401 Unauthorized.

Aktivita kontroly dostupnosti nové verze aplikace pošle serveru požadavek na porovnání verzí klientské aplikací a aplikace na serveru. S tímto požadavkem také pošle aktuální strukturu aplikace. Táto struktura bude obsahovat jména všech souborů aplikace, případně i s číslem verze souboru pokud je u souboru určena a relativní cestu ke každému souboru vzhledem ke kořenové složce aplikace. Jako odpověď se ji vrátí informace o tom zda-li nová verze je dostupná nebo není. V případě že je dostupná nová verze, příjde spolu s odpovědi i informace o tom, které soubory jsou změněné, které jsou nové a které byly odstraněny.

Aktivita stahování souborů pošle serveru požadavek na stáhnutí souboru s parametrem, který identifikuje konkrétní soubor. Server pošle zpět konkrétní soubor. Táto činnost se opakuje, dokud nejsou staženy všechny požadované soubory.

Kromě těchto základních případů může ještě nastat kdykoli nějaká chyba. Pokud dojde na serveru k nějaké chybě, která zapříčiní že server nebude moci dokončit požadovanou operaci, pošle server jako odpověď status kód 503 Service Unavailable.

Z předchozího je vidět, že komunikace obsahuje tyto druhy informací:

- Požadavek – krátký řetězec, identifikující konkrétní požadavek klientské aplikace.
- Status kód odpovědi – číslo identifikující stav odpovědi.
- Odpověď – krátký řetězec, vyjadřující jednoduchou odpověď serveru.
- Upřesňující informace – informace, přiložené k požadavku nebo odpovědi, nutné pro vykonání konkrétní činnosti, sloužící jako vstupní nebo výstupní data.

Pro výměnu zpráv se serverem bude většinou stačit použití HTTP hlaviček. Pro posílání upřesňujících informací, ale hlavičky stačit nebudou. Zde se nabízí použití XML formátu. Pro posílání XML zpráv přes HTTP je vhodné použít například komunikační protokol SOAP.

4.2.3 Porovnávání verzí aplikace

Při rozhodování o dostupnosti nové verze aplikace je potřeba porovnat verzi aplikace u klienta s verzi aplikace na serveru. Jak vyplývá z jednoho požadavku na systém, má být umožněno stahovat pouze změněné nebo nové soubory aplikace ze serveru, ne celou aplikaci. Aby bylo možné splnit tento požadavek, je nutné porovnávat každý soubor aplikace zvlášť. Nestačí tedy pouze porovnat čísla verze celé aplikace. Teprve při porovnání všech souborů aplikace lze zjistit, které soubory byly změněné, které jsou nové a které byly odstraněny. Pokud se při porovnávání verzí neobjeví žádný rozdíl, pošle server klientovi pouze informaci o tom, že není dostupná nová verze. Pokud se však na nějaký rozdíl narazí, je potřeba každý takový rozdíl zaznamenat a poslat klientovi. V postatě lze najít tři různé druhy rozdílů.

- Kontrolovaný soubor u klienta chybí: jde o nový soubor, který je potřeba stáhnout.
- Kontrolovaný soubor u klienta přebývá: jde o starý soubor, který je potřeba u klienta smazat.
- Kontrolovaný soubor u klienta je jiný než ten na serveru: jde o změněný soubor, který je potřeba stáhnout.

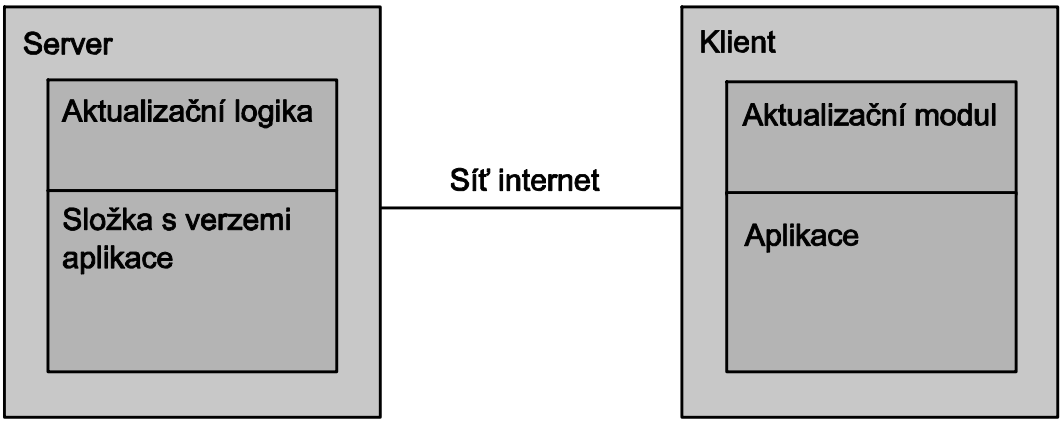
Jak porovnávat jednotlivé soubory? Jedna z možností je porovnávat názvy dvou souborů. V názvu souboru může být obsažena i verze souboru. Pro jednoduché a účelné porovnání postačí porovnat názvy dvou souborů na rovnost. Další možnost, kterou lze provést pouze u souborů typu JAR je načíst manifest a v něm vyhledat informace o verzi. Protože však ne každý JAR soubor obsahuje informaci o verzi, nebude to vždy spolehlivé. Další možnosti je vygenerovat pro každý soubor specifický kontrolní součet (hash kód) a po té porovnávat tyto kontrolní součty. Třetí způsob je nejjistější jak rozpoznat změny ve verzích aplikace, ale zároveň nejsložitější. Je potřeba specielních algoritmů, které mají vyšší nároky na procesor. Protože nebylo určeno jak rozpoznávat nové verze souborů, vystačíme si v našem systému s kombinací prvních dvou možností. Při porovnávání verzí souborů tedy budeme nejdříve porovnávat jejich názvy a pouze v případě JAR souborů, pokud budou názvy stejné se pokusíme najít a porovnat jejich verze, pokud budou uvedené.

4.3 Návrh systému

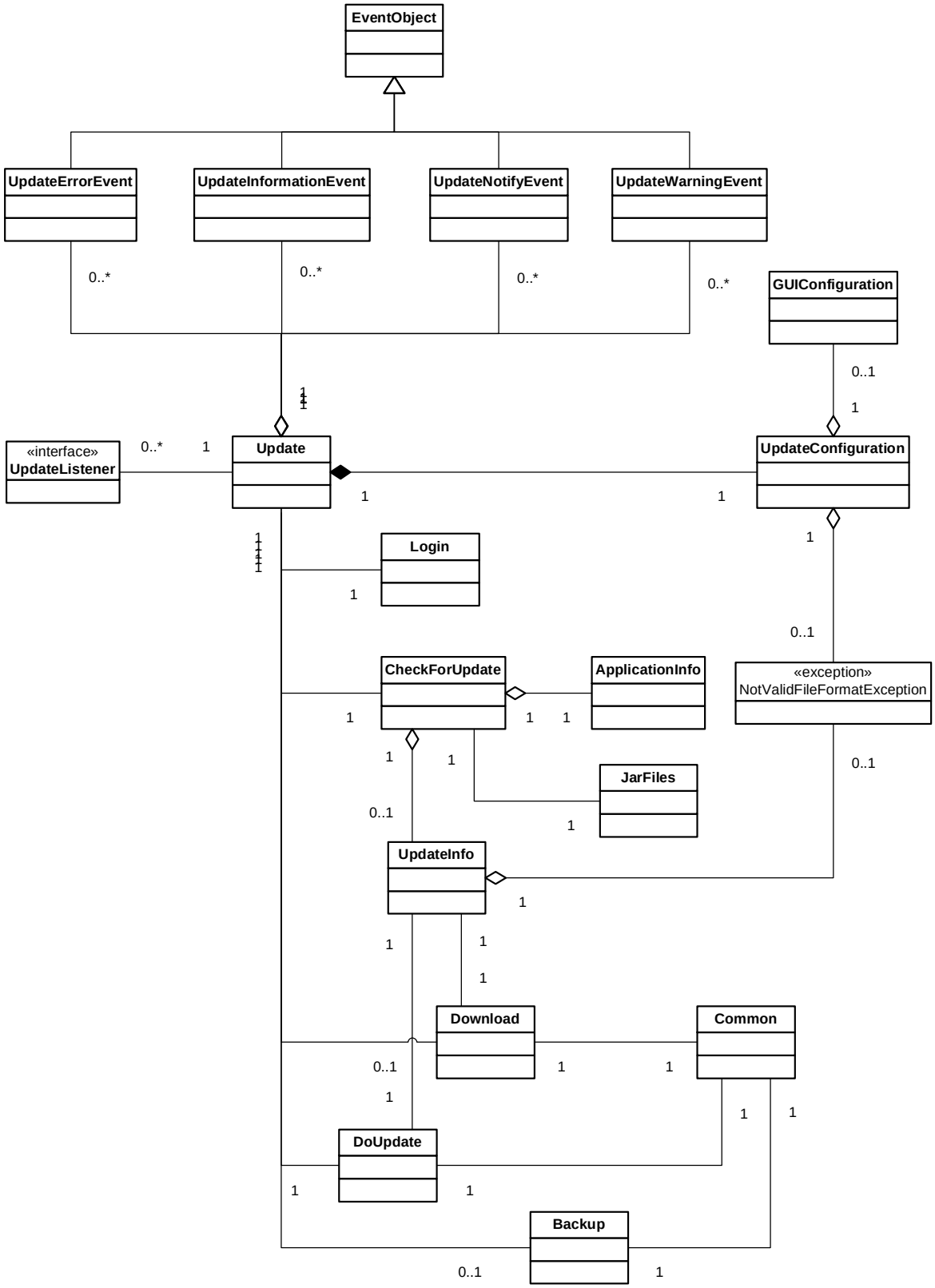
4.3.1 Struktura systému

Systém automatických aktualizací bude rozdělen na dvě základní části. Server a klient. Server bude poskytovatel nejnovější verze konkrétní aplikace a implementuje část aktualizačního procesu. Server bude provádět například kontrolu licenčních klíčů a porovnávat verze jednotlivých souborů. Klient

pak bude představován samotnou aplikaci doplněnou o aktualizační modul řídicí proces aktualizace. Aktualizační modul bude implementován jako knihovna. Táto knihovna se bude skládat z jednoho případně několika JAR souborů a jednoho konfiguračního souboru. Bude implementována jako samostatně pracující na aplikaci nezávislý celek. To by mělo zajistit možnost implementace do kterékoli Java aplikace.



Obr. 7. Struktura systému automatických aktualizací



Obr. 8. Diagram tříd aktualizačního modulu

4.3.2 Chování systému

Jelikož v zadání práce není požadována aktualizace bez nutnosti restartu, náš aktualizační modul se bude chovat podobně jak je popisováno v podkapitole 2.3.3. Jeho činnost bude zahájena na začátku aplikace a uživatel si může zvolit jednu ze dvou možností, aktualizovat na začátku a restartovat aplikaci nebo odložit aktualizaci na konec aplikace.

První tři aktivity se provedou zcela automaticky hned po spuštění aplikace. Při provádění všech činností aktualizace bude zobrazeno okno s informacemi o právě prováděné činnosti. Jak mile budou staženy všechny potřebné soubory pro vykonání aktualizace do dočasné složky, bude zobrazen dotaz na uživatele s výběrem dalšího postupu. Uživatel zvolí buď okamžitou aktualizaci nebo odložení aktualizace. Pokud uživatel zvolí okamžitou instalaci provede se nejdříve, pokud je tak nastaveno, záloha stávající verze aplikace. Zda-li provádět zálohu před instalací nebo ne bude nastaveno v konfiguračním souboru. Po provedení zálohy se spustí vlastní aktualizační proces. Na konci procesu se zobrazí hláška o nutnosti restartovat aplikaci. Po jejím odsouhlasení se aplikace ukončí. V případě že uživatel aktualizaci odloží na konec, spustí se aplikace ve stávající verzi a teprve při jejím ukončování se provede aktualizace na verzi novou. V tomto případě se žádost o restart nezobrazí. Pokud by během aktualizace nastala chyba, pokusí se modul obnovit aplikaci do předchozího stavu. Ovšem jen v případě, že je povoleno provádění zálohy před aktualizací.

Kromě spuštění celého procesu jak je popsáno výše bude umožněno samostatně spustit zálohu aplikace a také obnovení aplikace ze zálohy. Uživatel se tak bude moci vrátit k předchozí verzi aplikace, pokud by mu nová verze nevyhovovala. Jednotlivé aktivity budou implementovány odděleně. Tím bude modul umožňovat tvůrci aplikace různé možnosti implementace. Pořadí aktivit bude muset být zachováno, ale okamžik spuštění si může tvůrce vybrat sám. Tímto může celý proces aktualizace implementovat tak, aby uživatel aplikace o aktualizaci vůbec nevěděl nebo naopak může nechat potvrzovat jakoukoli akci uživateli. Na tvůrci aplikace bude i volba, kdy spouštět aktualizační proces. Modul mu totiž umožní třeba naplánovat provádění kontroly na novou verzi opakovaně v určitém časovém intervalu.

Jako v každém softwarovém díle, tak i v tomto systému může docházet k různým chybám. Nemusí to být přímo chyby konstrukční, ale chyby způsobené okolnostmi. Například může aplikaci spustit uživatel s omezenými právy a při stahování souborů ze serveru na počítač klienta může dojít k chybě při ukládání souboru na pevný disk. Uživatel nebude mít právo zápisu. Nebo může dojít k výpadku serveru a klient se nebude moci ze serverem spojit. V takovém případě je třeba reagovat na vzniklé situace a zvolit co se bude provádět dále. Aby tvůrce aplikace při implementaci tohoto modulu mohl na takovéto situace reagovat a dát tak případně uživateli na výběr co zvolit, je potřeba aby se o těchto situacích tvůrce nějak dozvěděl. Pro tyto potřeby bude aktualizační modul vybaven systémem vytváření událostí. Tento systém funguje na principu zdroje a posluchače. Zdroj vytvoří událost a informuje své posluchače o této události. Posluchač pak reaguje na tuto událost a vykoná určitou činnost. Aktualizační modul bude tedy zdrojem událostí a aplikace jeho posluchačem. Toho lze s výhodou využit nejenom v případě chyb, ale i v případě ukončení každé aktivity. Systém generování událostí takto vyřeší situace, kdy je potřeba dát vědět aplikaci co se stalo. Tvůrce aplikace pak na tyto události může jednoduše reagovat určitou činností. Aktualizační modul bude tedy generovat čtyři typy událostí.

- Informace – informace o tom že se zrovna něco provádí.
- Upozornění – informace o tom že se dokončila určitá aktivita.
- Varování – informace o nějakém problému, který není kritický (proces může pokračovat i tak, ale s určitým omezením).

- Chyba – informace o vzniklé chybě, pro kterou není možno pokračovat dále, dokud se nevyřeší.

4.3.3 Konfigurační soubor

Konfigurační soubor bude uložen v adresáři aplikace ve formátu XML. Tento soubor bude obsahovat specifické informace, které budou pro každou aplikaci jiné. Tyto informace bude mít uživatel nebo tvůrce aplikace možnost měnit podle svých vlastních potřeb. Jedna z informací, která bude v konfiguračním souboru uložena je umístění kořenové složky aplikace. Kořenová složka aplikace je adresář na pevném disku, ve kterém je aplikace nainstalována. Tento adresář tedy obsahuje všechny soubory aplikace. Kořenovou složku je potřeba znát proto, aby aktualizační modul věděl kde všude má hledat soubory aplikace, které se pak budou porovnávat s těmi na serveru. Dále pak je potřeba znát adresu serveru, ke kterému se bude klient připojovat. Tudíž táto adresa bude rovněž uložena v konfiguračním souboru. Z předchozího výkladu plyne, že uživatel bude mít možnost si volit zda chce provádět zálohu aplikace před aktualizací. Táto volba bude také součásti konfiguračního souboru. Bude zde uložen také licenční klíč, aby si jej mohla aplikace načíst vždy když bude potřeba přihlášení k serveru. Jelikož se budou nové a změnové soubory stahovat do dočasné složky a teprve potom se použijí při aktualizaci, je potřeba znát název a cestu k této složce. Takže i táto informace bude v konfiguračním souboru.

Všechny tyto informace budou v souboru XML uloženy jako pár jméno a hodnota. Kromě kořenového elementu budou tedy v souboru XML ještě další elementy. Každý element bude představovat jednu uloženou informaci. Jméno a hodnota pak budou uloženy v podobě dvou atributů každého takového elementu.

Jedena informace v souboru tedy může vypadat takto:
<backupBeforeUpdate value="yes" name="backupBeforeUpdate"/>
Jedná se o informaci zdali provádět zálohu před aktualizaci nebo ne.

4.3.4 Informace o aplikaci

Abychom mohli porovnávat jednotlivé verze aplikace na úrovni souborů je potřeba nejprve vytvořit strukturu aplikace jak na straně klienta, tak na straně serveru. Struktura aplikace bude používat opět XML syntaxi. V XML struktuře aplikace se budou vyskytovat, kromě kořenového, dva typy elementů. Typ „directory“ bude představovat adresář v aplikaci a bude mít jeden atribut. Tento atribut „name“ bude představovat jméno adresáře na disku. Každý element „directory“ může obsahovat opět tentýž element a navíc element „file“. File je druhý typ elementu představující soubor aplikace. Typ „file“ bude mít tři atributy. Atribut „name“ bude představovat jméno tohoto souboru, atribut „version“ bude představovat verzi souboru v případě že bude existovat a atribut „id“ bude představovat unikátní číslo souboru. Z těchto dvou elementu bude při každé potřebě vykonání kontroly na novou verzi aplikace vystavěna struktura aplikace, která umožní porovnání verzí jednotlivých souborů.

4.3.5 Informace o aktualizaci

V případě, že server při porovnávání dvou verzí aplikace narazí na rozdíly, je třeba tyto rozdíly zaznamenat a poslat klientovi. Ten na základě těchto informací učiní potřebné kroky pro vykonání aktualizace. Pro zaznamenávání těchto rozdílů s výhodou opět využijeme XML formát zápisu. Tentokrát budeme krom kořenového elementu používat jen jeden element pro uložení informací o rozdílech verzí. Bude to element „action“. Tento element bude mít krom jiných atribut „name“, který

bude označovat konkrétní akci, kterou je potřeba vykonat. Pro potřeby aktualizace si vystačíme se třemi akcemi. Akce se jménem „add“ bude říkat že daný soubor je nový a je potřeba jej do aplikace přidat. Akce se jménem „update“ bude říkat že daný soubor byl změněn a je potřeba jím nahradit stávající soubor. A poslední akce se jménem „delete“ bude říkat, že daný soubor již v nové verzi neexistuje a je potřeba jej na straně klienta smazat.

Další atributy elementu action jsou:

- Atribut „file-name“ – jméno souboru, kterého se daná akce týká.
- Atribut „relative-path“ – relativní cesta umístění souboru vztažená ke kořenové složce aplikace.
- Atribut „id“ – unikátní číslo souboru.

Tyto informace budou tedy obdrženy klientskou aplikaci jako odpověď ze serveru po provedení kontroly na dostupnost nové verze, v případě že nová verze bude existovat. Po přijetí se informace uloží do XML souboru na pevný disk pro pozdější využívání.

4.3.6 Práce s XML soubory

Z předchozích podkapitol je vidět, že se v systému bude docela často pracovat se strukturou XML. XML syntaxe bude použita v některých souborech uložených na pevném disku a také ve zprávách použitých při výměně informací se serverem. Aby se nám s XML formátem jednodušeji pracovalo, lze využít technologie JAXB, která podporuje práci s XML soubory v jazyce Java. Táto technologie umí kromě jiného převádět textovou reprezentaci XML formátu na Java objekty a naopak. Práce s touto technologii je jednoduchá. Pro každou třídu, která se bude používat ke tvorbě objektu nebo se objekt z ní vytvořený bude zapisovat do souboru, je nutno nejprve vytvořit schéma XSD. Toto schéma popisuje strukturu XML souboru se kterým se bude pracovat. Na základě takto vytvořeného schématu pak JAXB vygeneruje sadu tříd určených pro práci s takovým XML souborem.

4.3.7 Základní rozvržení tříd aktualizační knihovny

Nejdůležitější třídou bude třída „**Update**“. Ta bude představovat vstupní bod knihovny. V této třídě budou všechny základní metody představující jednotlivé aktivity procesu. Táto třída bude také zdrojem událostí, na které bude potřeba reagovat v průběhu procesu aktualizace. Základem implementace aktualizační knihovny v každé aplikaci tedy bude vytvoření instance této třídy. Na této instanci se pak budou volat jednotlivé metody zastupující aktivity procesu v určitém pořadí.

Další třídy knihovny by mohly představovat jednotlivé aktivity (akce) aktualizačního procesu.

- **Login** – táto třída se bude starat o činnosti spojené s přihlašováním k serveru
- **CheckForUpdate** – třída, která bude zajišťovat kontrolu nové verze aplikace
- **Download** – třída, která bude provádět stahování souborů ze serveru
- **DoUpdate** – třída, která provede samotnou aktualizaci aplikace
- **Backup** – třída, která bude mít na starosti zálohu aplikace a obnovení ze zálohy

Dále bude aplikace obsahovat třídy představující konfiguraci, strukturu aplikace, informace o aktualizaci. Tyto třídy budou přímo pracovat s XML soubory. Do jednotlivých tříd budou také rozděleny typy událostí, které bude generovat třída knihovna. Jedna třída by mněla zajistit grafické rozhraní pro práci s konfiguračním souborem. Uživateli tak bude zjednodušena práce při konfiguraci knihovny. Kromě těchto tříd bude potřeba také několika „pomocných“ tříd. Budou to třídy, které budou zajišťovat základní opakované operace.

Samotná aplikace pak může obsahovat některé třídy, podporující implementaci knihovny do aplikace. Rozvržení těchto tříd pak bude záležet na tvůrci aplikace.

4.3.8 Základní rozvržení tříd serverové části

Server bude obsahovat dva servlety. Jeden servlet bude obsluhovat stahování souborů při aktualizaci a druhý servlet se bude specializovat na obsluhu SOAP zpráv.

Třída zastupující servlet pro obsluhu stahování souborů se bude jmenovat „DownloadHandler“. Táto třída při požadavku o stáhnutí některého ze souboru aplikace ověří zdali je klient přihlášen (má platný licenční klíč) a pokud ano pošle mu daný soubor.

Třída zastupující servlet pro obsluhu SOAP zpráv se bude jmenovat „SOAPProcessor“. Táto třída upraví žádost zaslanou klientem pro vnitřní potřeby. Extrahuje SOAP zprávu, konkrétní požadavek a předá je třídě „RequestMessageProcessor“ ke zpracování. Od této třídy pak příjme SOAP odpověď, kterou připraví pro zpětné odeslání a zašle zpět klientovi. Také bude ověřovat, přihlášení klienta. Pokud klient nebude přihlášen, vždy jej přesměruje do sekce pro přihlášení.

Třída „RequestMessageProcessor“ pak bude vyhodnocovat požadavky klienta a na základě vyhodnocení posílat příslušným třídám na zpracování. Třída „ResponseMessageCreator“ bude vytvářet SOAP zprávy pro odpovědi serveru posílané klientům.

Třída „Keys“ bude obstarávat správu licenčních klíčů. Bude umožňovat přidávat a odebírat klíče ze seznamu. Třída „Login“ pak bude zajišťovat přihlašování klientů. Licenční klíč přijatý od klienta ověří proti seznamu licenčních klíčů na serveru a rozhodne zdali je platný. Podle tohoto rozhodnutí pak umožní nebo neumožní klientovi přístup na server.

Třída „AppDifferencesComparator“ bude porovnávat strukturu aplikace klienta se strukturou aplikace na serveru. Nejdříve však vytvoří strukturu nejnovější verze aplikace na serveru a po té ji porovná s tou, kterou přijala od klienta. Nalezne-li rozdíly ve strukturách, vytvoří informační zprávu, kterou pak pošle klientovi jako odpověď.

Kromě těchto tříd bude ještě potřeba několik pomocných tříd pro podpůrné operace.

5 Implementace systému pro automatické aktualizace

5.1 Základní informace

Aktualizační modul je tvořen dvěma JAR soubory napsaných v jazyce Java ve verzi 6. Tyto dva soubory zajišťují téměř veškerou činnost spojenou s aktualizacemi. Malá část činností je potom součásti aplikace, která modul používá.

Soubor „**AutomaticUpdate_DataModel.jar**“ obsahuje základní třídy pro práci se soubory XML a slouží jako podpora druhému JAR souboru, který je jádrem aktualizačního modulu. Technologie XML je použita pro ukládání některých vnitřních informací aktualizačního modulu na pevný disk a také pro přenos informací na server a zpět. Pro práci s XML soubory používá modul technologii JAXB. Táto technologie převádí textový XML formát na Java objekty a opačně, čímž zjednodušuje práci s XML soubory. Pro implementaci není nutno znát podrobnou činnost tohoto JAR souboru.

Soubor „**AutomaticUpdate_Client.jar**“ je jádro aktualizačního modulu. Obsahuje všechny potřebné třídy pro komunikaci se serverem a propojení s aplikací, která jej používá. Jednotlivé oblasti činností jsou rozděleny do několika balíků.

5.2 Přehled balíků, tříd a metod

Tento přehled se bude týkat JAR souboru AutomaticUpdate_Client.jar, jelikož je tento jádrem aktualizačního modulu a je potřeba jej znát při implementaci do určité aplikace. Podrobný popis všech tříd a metod umožní efektivnější využití při implementaci modulu do aplikace.

5.2.1 Balíky a třídy souboru AutomaticUpdate_Client.jar

Balíky „**automaticUpdate.applicationInfo**“, „**automaticUpdate.updateInfo**“ a „**automaticUpdate.updateConfiguration**“ obsahuji třídy, které „propojují“ tento JAR soubor s JAR souborem „**AutomaticUpdate_DataModel.jar**“ při práci s XML soubory. Obsahují metody pro zápis objektu do souboru, pro čtení objektu ze souboru, vytvoření nového souboru a převod objektu na DOM formát použitý v SOAP zprávách. Balík **automaticUpdate.updateConfiguration** navíc obsahuje třídu „**GUIConfiguration**“, která se používá pro změnu konfigurace aktualizačního modulu. Představuje grafické rozhraní pro komunikaci s uživatelem.

Balík „**automaticUpdate.updateExceptions**“ obsahuje třídu představující výjimku neplatného formátu souboru. Ta je vyvolána třídami pracujícími se soubory XML v případě, že načítaný soubor není platný XML soubor.

Balík „**automaticUpdate.internalSupport**“ obsahuje dvě třídy pro podpůrné činnosti:

- Třída „**JarFiles**“ slouží pro práci s JAR soubory. Načítá a vytváří instance archívu JAR, extrahuje Manifest a umí přečíst implementační verzi daného archívu. Toho se využívá při porovnávání verzí jednotlivých JAR archívů. Při zjišťování verze JAR archívu tedy hledá atribut „**Implementation-Version:**“.
- Třída „**Common**“ obsahuje funkce pro práci se soubory a adresáři. Kopíruje soubory, vytváří adresáře, kontroluje cesty k souborům, maže soubory a adresáře. Pokud se nepodaří smazat nějaký soubor či adresář, vytvoří seznam těchto selhání a zapíše jej do souboru **to-do.xml**. Tento soubor je při startu aplikace načten a pokouší se znovu smazat to, co před tím nešlo.

Balík „**automaticUpdate.updateEvents**“ obsahuje třídy představující jednotlivé události, které modul vytváří při své činnosti. To znamená, že aplikace, která implementuje aktualizační modul, se může dozvědět o všech zásadních činnostech, které nastanou, prostřednictvím těchto událostí. Je potřeba pouze implementovat rozhraní „**UpdateListener**“, a zaregistrovat jednu ze svých tříd jako posluchače událostí.

- Třída „**UpdateListener**“ – rozhraní posluchače událostí aktualizačního modulu. Obsahuje čtyři metody, které je třeba definovat ve třídě implementující toto rozhraní. Každá metoda odpovídá jednomu typu události.
- Třída „**UpdateInformationEvent**“ – událost, která informuje o něčem co se zrovna děje. Většinou počátek nějaké činnosti. Má pouze informativní účel, například pro výpis prováděných činností na obrazovku. Nese informaci o objektu, který ji vyvolal a popis toho co se stalo.
- Třída „**UpdateNotifyEvent**“ – upozorňuje na události. Nese informaci o objektu, který ji vyvolal, popis toho co nastalo a typ události. Většinou upozorňuje na právě dokončenou akci s jejím výsledkem. Pomocí tohoto typu událostí se řídí chod činností aktualizačního procesu tím, že se na tyto událostí vhodně reaguje v aplikaci.
- Třída „**UpdateErrorEvent**“ – představuje závažné chyby vzniklé při činnostech aktualizace. Nese informaci o objektu, který ji vyvolal, popis toho co se stalo, typ chyby a pokud existuje, informaci o výjimce která nastala. Rovněž pomocí tohoto typu události, lze řídit chod činností při aktualizacích vhodnou reakcí aplikace.
- Třída „**UpdateWarningEvent**“ – představuje méně důležité chyby či problémy, které nastanou v průběhu činností aktualizace. Nese informace o objektu, který ji vyvolal, popis toho co se stalo, typ problému a pokud existuje, informace o vzniklé výjimce. Jelikož varování představuje chyby nebo problémy, které přímo nebrání v pokračování procesu aktualizace, není nutno vždy na ně reagovat.

Hlavní tok činností je řízen reakcí na události „**Notify**“ a „**Error**“. Tyto událostí dávají vědět, kdy se co dokončilo a jak, případně kdy nastala chyba. Jednotlivé typy těchto událostí slouží k rozpoznání toho co se vlastně stalo.

Tak například jeden z typů chyby je „**LOGIN_FAILED**“. Způsob řízení toku činností spočívá v tom, že pokud při přihlašování se k serveru nevznikne chyba, je vyvolána událost „**Notify**“ typu „**DONE**“. V tomto případě se může pokračovat v dalším kroku. Pokud však dojde k chybě, je vyvolána chyba typu „**LOGIN_FAILED**“ zmíněná výše. V tomto případě modul nemůže pokračovat dále a je na vývojáři, co se rozhodne udělat dál. Buď kontrolu na novou verzi aplikace přeruší a spustí samotnou aplikaci nebo se pokusí zopakovat přihlášení.

Balík „**automaticUpdate.updateActions**“ obsahuje třídy, představující jednotlivé akce (aktivity) prováděné při kontrole aktualizací.

- Třída „**Login**“ provádí přihlašování k serveru. Nejdříve odešle na server požadavek na autorizaci a předá serveru licenční klíč, což je řetězec znaků. Server ověří licenční klíč a pošle třídě **Login** informaci o úspěchu či neúspěchu přihlášení. V případě úspěchu i „**sessionid**“, které třída **Login** uloží pro další použití. Přihlašování na server první krok, který je třeba vykonat.
- Třída „**CheckForUpdate**“ provádí kontrolu nové verze aplikace. Nejdříve vytvoří strukturu současné verze aplikace (seznam adresářů a souborů případně i s verzí). Po té převede tuto strukturu na XML zprávu a tu pošle na server s požadavkem na provedení porovnání struktury se strukturou na serveru. Server odpoví na tento požadavek buď jednoduchou informací o shodě struktury (nejsou žádné změny)nebo, v případě rozdílů (nová verze), pošle seznam s akcemi, které je třeba vykonat (přidat, změnit či smazat). Třída **CheckForUpdate** odpověď zpracuje, uloží seznam akcí jednak do souboru na disk a jednak do vnitřní proměnné a vytvoří

událost informující o tom zdali je nebo není dostupná nová verze aplikace. Kontrola nové verze aplikace je druhý krok, který je nutno vykonat. Tento krok je závislý na přihlášení k serveru.

- Třída „**Download**“ provádí stahování potřebných souborů ze serveru. Pokud existuje seznam akcí, které se mají vykonat pro úspěšnou aktualizaci, načte jej a postupně provede stáhnutí souborů, které jsou označeny jako změněné a těch, které jsou označeny jako nové. Tyto soubory uloží do dočasné složky nazvané „**update**“. Stahování souborů je třetí krok v aktualizaci. Závisí na přihlášení k serveru a kontrole na změny.
- Třída „**Backup**“ provádí zálohu aplikace a obnovení ze zálohy. Pokud je v konfiguračním souboru aktualizačního modulu nastaveno provádět zálohu aplikace před aktualizací, je provedení zálohy čtvrtým krokem. Celá struktura aplikace („všechny“ soubory a složky) jsou zkopírovány do adresáře „**backup**“. V konfiguračním souboru lze zvolit adresáře nebo soubory, které se mají „vyjmout“ z toku činností. To znamená, že každý soubor či adresář, který je zapsán v poli „**Nevšímat si souboru či složek**“ konfiguračního souboru nebude aktualizován, zálohován ani obnovován. Prostě se těchto souborů či složek nebude aktualizační modul vůbec všímat. Pokud se nezdaří aktualizace, použije se táto třída také k provedení obnovení ze zálohy, pokud záloha existuje. Pro provedení obnovení aplikace ze zálohy tedy musí být v konfiguračním souboru nastaveno provádění zálohy před aktualizaci. Toto je pak posledním krokem v toku činností.
- Třída „**DoUpdate**“ provádí samotné nahrazení stávajících souborů za nové, případně přidává další soubory nebo maže ty nepotřebné. Vše záleží na seznamu akcí, které se mají provést. Tento seznam si táto třída načte a provádí postupně jednotlivé akce. Táto činnost je čtvrtým případně pátým krokem.
- Třída „**Update**“ je hlavní třídou celého modulu. Obsahuje metody, které volají předchozí popsané třídy a hlídá při tom jestli nenastala nějaká chyba. Táto třída je zdrojem událostí. Je to vlastně vstupní bod aktualizačního modulu. Použití aktualizačního modulu tedy kromě výše popsaných činností spočívá v tom, že se vytvoří instance této třídy (objekt) a postupně se volají jednotlivé metody této instance. Po vytvoření instance této třídy, je nutné zaregistrovat se jako posluchač událostí, jinak nelze s modulem aktualizací pracovat. Veškeré informace o chybách, výsledcích všech kroků a dokončení určité činností jsou totiž oznamovány prostřednictvím událostí.

5.2.2 Seznam metod třídy „Update“

Protože je táto třída nejdůležitější, je nutné popsat její metody.

Metoda „**verifyConfiguration**“ provádí základní ověření konfiguračního souboru aktualizačního modulu. Ověřuje, je - li uvedena kořenová složka aplikace a pokud ano tak jestli existuje. Dále ověřuje jestli jsou zadány jména pracovních složek, jestli je zadáno URL serveru na který se budou posílat požadavky a má - li toto URL platný formát a nakonec je - li zadán licenční klíč. V případě že nějaké nastavení není v pořádku vyvolá jako událost chybu typu „**BAD_CONFIGURATION**“ a vrací hodnotu „false“. Pokud je vše v pořádku, vyvolá událost upozornění typu „**DONE**“ a vrací hodnotu „true“.

Metoda „**login**“ volá metodu stejného jména třídy Login. Pokusí se přihlásit k serveru pomoci licenčního klíče a získat ze serveru session id. V případě že uspěje vyvolá událost upozornění typu „**DONE**“ a vrátí hodnotu „true“. V případě že se přihlášení nepovede vyvolá jednu ze dvou chybových událostí. Chyba „**UNAVAILABLE**“ je vyvolána v případě že je server nedostupný a chyba „**LOGIN_FAILED**“ je vyvolána v případě, že selže ověřování licenčního klíče. Jinými slovy licenční klíč je neplatný.

Metoda „**checkToDo**“ kontroluje zdali existuje soubor „to-do.xml“. V případě že ano, načte jej a zjistí, jestli se má provést nějaká činnost nebo ne. Pokud ano, provede ji. V současné implementaci se tento soubor používá pouze pro zápis souborů a adresářů, které se při mazání v době aktualizace nepodařilo smazat. Například byl li soubor v používání. Tudíž po načtení tohoto souboru se zkouší znovu smazat to, co se před tím nepovedlo. Pokud se soubor nebo adresář nepodaří opět smazat, vyvolá metoda varování typu „**UNABLE_TO_DELETE**“. Pokud se nepodaří načíst soubor „to-do.xml“ vyvolá metoda varování typu „**JAXB_ERROR**“. Pokud byla před touto metodou volána metoda „**verifyConfiguration**“ a vyvolala chybu „**BAD_CONFIGURATION**“, vyvolá metoda „**checkToDo**“ varování „**CAN_NOT_CONTINUE**“. Pokud se vše podaří a metoda dojde na konec své činnosti, vyvolá upozornění typu „**DONE**“. Táto metoda vytváří při nějaké chybě pouze událost varování. To z toho důvodu, že její činnost není kritická a aktualizace se dá dokončit i v případě selhání této metody.

Metoda „**checkUpdate**“ volá metodu „checkForUpdate“ třídy „CheckForUpdate“, která vytvoří strukturu aplikace a odešle jí na server s dotazem na kontrolu nové vrze. Táto metoda může vytvářet více událostí: Například varování – „FILE_NOT_FOUND“, „JAXB_ERROR“ nebo chyby typu „JAXB_ERROR“, „SOAP_ERROR“, „URL_ERROR“, „NOT_EXIST_ROOT_DIRECTORY“, „UNKNOWN_ERROR“. Pokud nevznikne žádná chyba, vrátí metoda „checkForUpdate“ hodnotu „true“ v případě, že je dostupná nová verze aplikace na serveru nebo hodnotu „false“ v případě že není dostupná. Pokud je nová verze dostupná, volá se dále metoda „**dovnloadNewFiles**“, která stáhne všechny potřebné soubory ze serveru do dočasného adresáře uvnitř kořenové složky aplikace. Název dočasného adresáře je dán v konfiguračním souboru. V tomto souboru bude taky uložen xml soubor „update-info.xml“, který obsahuje veškeré akce, které je potřeba vykonat k úspěšné aktualizaci. Při stahování souborů můžou být vyvolány tyto událostí: Chyby „IN_OUT_ERROR“, „URL_ERROR“, „UNAVAILABLE“ , NOT_AUTHORIZED“, „NULL_SESSIONID“, „FAILED“, varování „CAN_NOT_CONTINUE“ nebo při dokončení metody upozornění „UPDATE_AVAILABLE“, „NO_UPDATE“.

Metoda „**doUpdate**“ zjistí jestli se má nebo nemá provádět záloha aplikace před aktualizací a v případě že má, provede nejdříve zálohu aplikace tak, že zkopíruje celou strukturu aplikace do složky zálohy. Složka zálohy se nachází uvnitř kořenové složky aplikace a její název je dán v konfiguračním souboru. Dále provede samotnou aktualizaci. To znamená, že si načte soubor „update-info.xml“ a postupně provede všechny jeho akce. Při provádění zálohy se můžeme setkat s těmito chybami: „ACCESS_DENIED“, „NOT_EXIST_ROOT_DIRECTORY“ a varováním „CAN_NOT_CONTINUE“, „BACKUP_FAILED“. Při provádění aktualizace se můžeme setkat s těmito chybami: „JAXB_ERROR“, „FILE_NOT_FOUND“ a varováním „UNABLE_TO_DELETE“. Při úspěšném provedení aktualizace je vyvolána upozorňující událost „RESTART_NEEDED“. V případě že se během aktualizace vyskytne nějaká vážná chyba, je volána metoda „restoreFromBackup“ třídy „Backup“. Táto metoda se pokusí vrátit stav aplikace do stavu předchozího v jakém se aplikace nacházela před započetím aktualizace. Při provádění obnovy ze zálohy se mohou vyskytnout tyto událostí: Chyby „ACCESS_DENIED“, „NOT_EXIST_BACKUP_DIRECTORY“ a upozornění „RESTART_NEEDED“, „FAILED“. O obnovu ze zálohy se modul pokouší pouze v případě že je v konfiguračním souboru nastaveno provádění zálohy před aktualizací.

5.2.3 Konfigurační soubor

Na obrázku níže je ukázka konfiguračního souboru. Každá aplikace bude mít hodnoty atributů mírně odlišné, dle svých aktuálních potřeb.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<update-configuration xmlns="http://xml.netbeans.org/schema/update-configuration">
    <appRootDir value="D:\Aplikace\UniSave 2006 upr" name="appRootDir"/>
    <backupName value="backup" name="backupName"/>
    <configName value="config" name="tempName"/>
    <updateName value="update" name="updateName"/>
    <serverUrl value="http://79.127.240.50:8080/AutomaticUpdate_Server" name="serverUrl"/>
    <excludeDirectories/>
    <backupBeforeUpdate value="yes" name="backupBeforeUpdate"/>
    <licenceKey value="12345656" name="licenceKey"/>
</update-configuration>
```

Obr. 9. Ukázka konfiguračního souboru

- Atribut „**appRootDir**“ obsahuje absolutní cestu ke kořenové složce aplikace. K této složce jsou vztaženy všechny relativní cesty aktualizačního modulu.
- Atribut „**backupName**“ obsahuje jméno složky, ve které je ukládána záloha aplikace.
- Atribut „**configName**“ obsahuje jméno složky, ve které je uložen konfigurační soubor.
- Atribut „**updateName**“ obsahuje jméno dočasné složky, do které se stahují soubory nové verze aplikace před aktualizací.
- Atribut „**serverUrl**“ obsahuje adresu serveru, ke kterému se připojuje aktualizační modul.
- Atribut „**excludeDirectories**“ obsahuje adresáře a soubory oddělené čárkou, které mají být vyjmuty z toku činností při aktualizaci. To znamená, že každý soubor nebo adresář uvedený zde, bude aktualizačním modulem při všech jeho činnostech „přehlížen“. Nebude zálohován, aktualizován, kontrolován ani mazán. Táto vlastnost umožňuje mít v aplikaci soubory, které nejsou přímou součásti aplikace. Například výstupní soubory aplikace (data).
- Atribut „**backupBeforeUpdate**“ „říká“, jestli se bude provádět záloha aplikace před aktualizaci nebo ne. Možné hodnoty jsou „yes“ nebo „no“.
- Atribut „**licenceKey**“ obsahuje uložený licenční klíč.

Složky backupName, configName a updateName se nachází přímo v kořenovém adresáři aplikace a jsou to systémové složky aktualizačního modulu. Možnost změnit si názvy těchto složek je výhodná v případě již existující aplikace, ve které by mohly výchozí názvy těchto složek kolidovat s jinými složkami stejného názvu. Tyto složky jsou taktéž vyjmuty z toku činností. Složka „configName“ zcela, zbývající dvě složky částečně. Každopádně nejsou zahrnuty do struktury aplikace posílané na server k porovnání verzí.

6 Použití systému v aplikaci UniSave

Základní implementace systému automatických aktualizací do aplikace UniSave je provedena v pěti krocích.

6.1 Manifest

Aby aplikace UniSave věděla o přítomnosti aktualizačního modulu, je potřeba přidat odkaz na jeho dva JAR soubory do proměnné „Class-Path“ v souboru manifestu. Proměnná „Class-Path“ tedy bude mít o tyto dva řádky na víc:

- lib\AutomaticUpdate_Client.jar
- lib\AutomaticUpdate_DataModel.jar

6.2 Adresář „lib“

Do tohoto adresáře je potřeba přidat oba JAR soubory aktualizačního modulu. Jedná se o tyto soubory:

- AutomaticUpdate_Client.jar
- AutomaticUpdate_DataModel.jar

6.3 Třída „Run“

Třída „Run“ se nachází v balíku „**unisave2006**“ a je hlavní třídou aplikace. Táto třída obsahuje téměř celý kód pro řízení aktualizace protože je zde „začátek“ aplikace. Původní řádky obsažené v metodě „main“, které spouštěly aplikaci UniSave, byly umístěny do nové metody s názvem „**runApplication**“, která se volá až později, po provedení potřebných kroků aktualizace. Do metody „main“ bylo umístěno volání metody „**startUpdate**“. Metoda „startUpdate“ vytváří instanci třídy „Update“ aktualizačního modulu, která jak je výše popsáno je vstupním bodem aktualizačního modulu. Po vytvoření instance také zaregistruje třídu „Run“ jako jejího posluchače událostí. Dále je do třídy „Run“ přidáno několik řídících metod a také implementovány metody rozhraní „**UpdateListener**“. Byla také upravena původní metoda „**createAndShowGUI**“, která vytváří hlavní okno aplikace UniSave. Do této metody byl přidán záznam, který nastavuje v hlavním okně aplikace UniSave odkaz na instanci třídy „update“ aktualizačního modulu.

6.4 Třída „AutomaticUpdateStatus“

Táto třída byla přidána do aplikace za účelem vytvoření grafického uživatelského rozhraní pro komunikaci s uživatelem. Obsahuje výčtový typ pro identifikaci operací. Tento výčet při překladu vytvoří další třídu "AutomaticUpdateStatus$Operation". Tudíž se v balíku „unisave2006.gui“ objeví dvě nové třídy. Objekt této třídy vytvoří okno, které zobrazuje veškeré událostí generované aktualizačním modulem. Ve spodní části okna jsou ovládací tlačítka, která slouží pro interakci s uživatelem.

6.5 Třída „MainFrame“

Třída „**MainFrame**“ se nachází v balíku „unisave2006.gui“. Zde se nachází „konec“ aplikace, což je místo ve kterém je metoda, která ukončuje aplikaci v případě potřeby. Do této třídy byla přidána proměnná „**appUpdate**“, která udržuje referenci na instanci třídy „Update“. Dále byla přidána metoda

„**setUpdateReference“,** která nastavuje proměnnou „appUpdate“. Táto reference je důležitá při ukončení aplikace UniSave. Je potřeba aby se o ukončení dalo „vědět“ objektu třídy „Update“. Tento okamžik se totiž využívá v případě odložené aktualizace. Z tohoto důvodu byla také provedena změna v metodě „**exitApplication**“. Pro ukončení aplikace se nevolá „Systém.exit()“ jak se volalo původně, ale „setVisible(false)“ a „dispose()“, což neukončí celou aplikaci, ale jen hlavní okno aplikace. Dále se dává oznámení o ukončení hlavního okna instanci třídy „Update“. Táto instance pak zkontroluje, jestli není připravena aktualizace. Pokud ano, provede ji a po té ukončí aplikaci. Pokud ne, pouze ukončí aplikaci.

7 Závěr

Cílem této bakalářské práce bylo navrhnout a naimplementovat systém automatických aktualizací pro aplikace v jazyce Java. Před započetím tvorby návrhu systému bylo potřeba nastudovat několik prací, týkajících se problematiky aktualizací obecně. Teprve potom bylo možno určit jednotlivé oblastí této problematiky a na tomto základě rozdělit systém do několika menších celků. Při hlubším studiu této problematiky vyšlo najevo, že některé oblastí jako například dynamické zavádění tříd za běhu aplikace nebo přenos stavu staré verze aplikace do nové, nejsou vůbec jednoduché a jejich aplikace do systému by vyžadovala mnohem více úsilí a delší časové období na provedení. Odměnou za toto úsilí by však byla podstatně širší použitelnost takového systému. Právě v těchto oblastech problematiky bych viděl námět na další téma bakalářské či diplomové práce, které by mohlo případně navazovat na tuto práci.

Při návrhu systému pak bylo nutno ještě nastudovat dvě technologie, které zjednodušily implementaci systému. Jedná se o technologii JAXB, která podporuje práci s XML soubory a technologii SOAP, která podporuje komunikaci dvou aplikací přes síť. Při implementaci samotné jsem se setkal s několika problémy, které se však podařilo úspěšně vyřešit. Při testování a ladění systému vyšlo najevo, že by se systém dal doplnit o další užitečně „věci“. V případě serverové části, by bylo přínosem implementovat rozšíření, které by zjednodušilo práci vývojáře při nahrávání nové verze aplikace na server a usnadnilo konfiguraci serveru. Dále by se dalo rozšířit porovnávání souborů při rozhodování o nové verzi aplikace nejen podle jména a čísla verze, ale také podle nějakého hash kódu. Celý systém by pak bylo vhodné rozšířit o určitý způsob zabezpečení. Jednak komunikaci mezi klientem a serverem a pak také ověřovat nové soubory po stažení ze serveru jako ochranu proti podvrhu. Tady bych viděl další možnost námětu na bakalářskou práci.

Při provádění testů aktualizačního modulu a simulování nejrůznějších chyb, které mohou nastat v reálném provozu se podařilo dosáhnout stabilního stavu, který končil vždy buďto úspěšným provedením aktualizace nebo zastavením procesu a ohlášením chyby. Při výskytu některých chyb se známou příčinou a jednoduchou opravou, je nabídnuta uživateli pomoci grafického rozhraní možnost úpravy konfigurace, která by danou chybu měla vyřešit. Z hlediska požadavků na systém a výše zmíněných testů byl tedy splněn cíl této práce.

Studium problematiky a tvorba systému automatických aktualizací mi byla jak teoretickým, tak praktickým přínosem. Dozvěděl sem se o některých, pro mě nových technologiích, pronikl hlouběji do procesu zavádění Java tříd virtuálním strojem a nabyl zkušeností s rozšiřováním „cizího“ kódu. Věřím, že táto práce bude přínosem i dalším lidem, zejména vývojářům Java aplikací.

8 Použitá literatura

[1] ŠIK, Petr. *Nahrávání tříd v Javě - Class Loading* [online].
Dostupné z: <http://www.bscsro.cz/articles/javaclassloader.pdf>

[2] PICHLÍK, Roman. *Za tajemstvím classloaderu* [online]. 27. 2. 2007
Dostupné z: <http://www.dagblog.cz/2007_02_25_archive.html>

[3] VANDEWOUDE, Yves: *Dynamically updating component-oriented systems*. [Disertační práce], Katholieke Universiteit Leuven, Leuven, 2007,
Dostupné z: <http://www.cs.kuleuven.be/publicaties/doctoraten/cw/CW2007_06.pdf>

[4] GEE, Steve: *Java Application Update Utility Software* [online]. 25. 7. 2005
Dostupné z: <http://sourceforge.net/projects/jauus>

[5] JELÍNEK, Lukáš: *Java – datové typy* [online]. 29. 11. 2005, č. 21
Dostupné z: <http://www.linuxsoft.cz/article.php?id_article=1025>

[6] KOSEK, Jiří: *Inteligentní podpora navigace na WWW s využitím XML* [online].
Dostupné z: <http://www.kosek.cz/diplomka/html/websluzby.html>

[7] Wikipedie – otevřená encyklopedie – česká verze. 18. 4. 2010
Dostupné z: <http://cs.wikipedia.org/wiki/SOAP>

[8] HEROUT, Pavel: *Java a XML*. 1. vydání. České Budějovice: KOPP, 2007.
ISBN 978-80-7232-307-4

9 Přílohy

Příloha č. 1 Zdrojové kódy

Veškeré zdrojové kódy systému se nacházejí na přiloženém CD v adresáři „**Zdrojové kódy**“.

Příloha č. 2 Ukázka vlastní implementace class loaderu

Java třída, představující vlastní class loader, se nachází na přiloženém CD v adresáři „**Zdrojové kódy/ CustomClassloader**“.

Příloha č. 3 Instalační a uživatelský manuál

Manuály a popis změn provedených v aplikaci UniSave při implementaci systému automatických aktualizací se nacházejí na přiloženém CD v adresáři „**Manuály**“.

Příloha č. 4 Elektronická forma této práce

Elektronická forma této bakalářské práce se nachází na přiloženém CD v adresáři „**Text Bc. práce**“.

Příloha č. 5 Literatura

Některá použitá literatura se nachází na přiloženém CD v adresáři „**Literatura**“.